Master’s Thesis in Computer Science
Uppsala University, 20 p
2007-03-27

Development of a Prototype for JDF Enabled Software

Ola Stering

2007-03-27

Abstract

The goal of this thesis project is to develop a prototype for an application that can act as a reference for other developers wishing to implement JDF-support in their software. It is required by CIP4 that the prototype is implemented in Java and supports CIP4’s Base ICS. The thesis project involves analyzing the problem domain, gathering detailed requirements, object-oriented modeling, and implementation. Emphasis is placed on a modular application design and an iterative development approach. This thesis explains the design and architecture including programming decisions for such a software application. It provides a detailed overview of the Elk project, its design and implementation strategies.

Keywords: JDF, software development, print production workflow.

Supervisors: Claes Buckwalter, Rainer Prosi
Examiner: Mikael Pettersson

Table of Contents

List of figures

1. Introduction

1.1. Problem statement

The trend in the printing industry is towards an increasing amount of individual print jobs and shorter run lengths. In order to maintain profit print shops need to efficiently manage the growing amount of administrative information and streamline their print production workflow. Information interchange between, and integration of, administrative systems and production equipment is crucial in order to be able to optimize the workflow and maintain profitability.

The *Job Definition Format (JDF)* is a joint effort by the vendors in the printing industry that attempts to solve the problems described above. JDF specifies an XML-based data interchange format for integrating systems, software and equipment in a print production workflow. In order to help vendors implement JDF-enabled software *CIP4* (see below), the organization responsible for JDF, would like to provide a reference implementation of a JDF device. In August 2004 work began on a reference implementation, codenamed Elk.

The CIP4 required the following tasks to be completed:

- Provide a fully functional JDF Device conforming to the Base ICS [5]
- Analyze and describe the software design of Elk

A more detailed description of problems arisen and how they were solved can be found in 11 Appendix C: The author's programming contributions

1.2. Goal

The goal of this thesis was to further develop the Elk according to the requirements from CIP4. Also, a goal is to explain the architecture and the software design approaches used in Elk; the definition of the requirements of building a JDF enabled software device as well as explaining the concrete usage of design patterns and programming solutions. Another goal is to provide a detailed overview of the Elk project, its design and implementation strategies.

1.3. Method

In order to contribute in a meaningful way in constructing a JDF Device research and reading in the JDF Specification [27] and other printing industry related literature was conducted. Involvement in the technical workgroup within CIP4 that focus on tools and infrastructure also helped the author to get acquainted with JDF and the related work. Basic understanding of XML [37][41] was also of importance. Since the *Elk Project* development started in 2004 the author needed to fully understand the concepts and the architecture of the framework to be able to further develop it. The chapters "The Elk Framework" and the "The Elk Reference Implementation" explains these concepts. The Elk Framework development process also included reading and understanding of testing using JUnit [34], version and bug report system JIRA [32] as well as the logging concepts of log4J [39]. Design choices in the development work has been made in discussion with, among others Claes Buckwalter and by the author's previous knowledge of program development combined with problem specific literature and the use of online resources. The UML diagrams in this thesis follow the suggestions in UML Distilled [19].

1.4. Conventions

Terms that are encountered for the first time and later in the thesis are explained further are given in *italics*. Examples: *JDF, CIP4, Device*. A brief explanation of some of the words in italic may also be found in Appendix B: Glossary.

Terms directly derived from the JDF Specification [27] starts with a Capital letter. Examples: Device, Agent, Controller, MIS.

Java classes and methods are written with the `code` font as well as console calls and filenames. Examples: `SubscriptionManager, MemoryQueue, IncomingJMFDispatcher.`

In cases where classes with long names are referred to several times in the same section, the abbreviation used for the rest of the section is given within parenthesis. Example: The `SimpleAsyncSubscriptionManager (SubManager)` is often used here. The `SubManager` is also….

1.5. Acronyms

Some of the acronyms that are used in this thesis:

ERI	The Elk Reference Implementation
XML	Extensible Markup Language
ESB	Enterprise Service Bus
MIS	Management Information System
JDF	Job Definition Format
ICS	Interoperability Conformance Specification
JMF	Job Message Format
HTTP	Hyper Text Transport Protocol
URL	Uniform Resource Locator

Most of these acronyms are explained more thoroughly later in the thesis.

1.6. Disposition

First, the reader is introduced to CIP4 and JDF. Some of the terminology which will be used throughout the paper is also defined. The next chapter starts out with explaining the requirements and generic functionality of a JDF enabled software prototype which the Elk framework is built upon. Next, the Elk Project is introduced to the reader and the Framework as well as the reference implementation is explained. The following chapter explains the program design and the development process of the project including discussions and programming specific solutions to arisen problems. Finally, the resulting software is discussed and some thoughts on future and related works are given. The appendices include some help on how to run Elk; a glossary and the author's programming contributions to the project.

2. JDF and CIP4

In order to understand the architecture and design of Elk the reader needs to be acquainted with JDF and its structure. A brief summary of CIP4 and JDF is given in the following sections. More comprehensive information can be found at cip4's web site [56] and in the JDF Specification [27].

2.1. CIP4

The International Cooperation for the Integration of Processes in Prepress, Press and Postpress Organization [56] is a non-profit organization and was formed in September 2000. CIP4 brings together end-users, vendors and consultants in the Graphic Arts industry to unite and develop standards, tools and overcome communication barriers within the industry.

The need to streamline and increase efficiency in a print production workflow becomes more and more prominent for each day that pass. The manufacturing industry has for a long time been able to automate its processes, and the Graphic Arts industry has been far behind. The first great effort to automate the processes in a print production workflow was made by CIP3, the predecessor to CIP4. CIP3 and its members developed the Print Production Format (PPF) which had some success in postpress operations. Yet another early attempt was made by Adobe and its Portable Job Ticket Format (PJTF) which was a method for exchanging print metadata. Other attempts were Graphic Communications Association's Industry Architecture Project and IFRA's ifraTrack.

With these formats in mind, an XML-based job ticket format was created by the companies Adobe, Agfa, Heidelberg and MAN Roland which they called the Job Definition Format (JDF). They asked CIP3 to take over the stewardship of the specification and to make JDF an open, public for all, data interchange format. The CIP3 merged into CIP4 which now manages the specification.

Not only does CIP4 manage the specification of JDF, they also find new user requirements, develop and enhance it. They also maintain and develop a Software Development Kit (SDK) and provide applications and tools for building JDF software. CIP4 conducts its work through technical working groups each focusing on some part of the specification. Some examples are the Prepress, Digital Printing and Tools & Infrastructure groups.

2.2. JDF

This subsection introduces JDF to the unfamiliar reader. This summary is very general and the author refers to the JDF 1.2 Specification [27] and the ICS Documents [28] for a complete overview. The author believes that the concepts explained in this section are the most important in order to understand the contents of the remaining parts of this thesis.

"Because of workflow system construction in today's industry, the principal subsection procedures of a printing job—prepress, press, and postpress—remain largely disconnected from one another. JDF provides a solution for this lack of unity. With JDF, a print job becomes an interconnected workflow that runs from job submission through trapping, RIPing, filmmaking, platemaking, inking, printing, cutting, binding, and sometimes even through shipping."
([27], section 2.2 JDF Workflow)

The above sentence clearly overviews the possibilities and aim of JDF. The emphasis lies on the interaction and information interchange between applications, machines and humans. XML is a descriptive language which has syntactic rules which enables computers to interpret and verify contents and at the same time it is easily understood by humans. Therefore, XML is used to structure and define the products and processes in JDF. In order to follow a job defined in JDF through the execution process, constructs for auditing and messaging are provided. The messaging format is called *Job Message Format* (JMF, see below). The figure below overviews the uses of JDF:

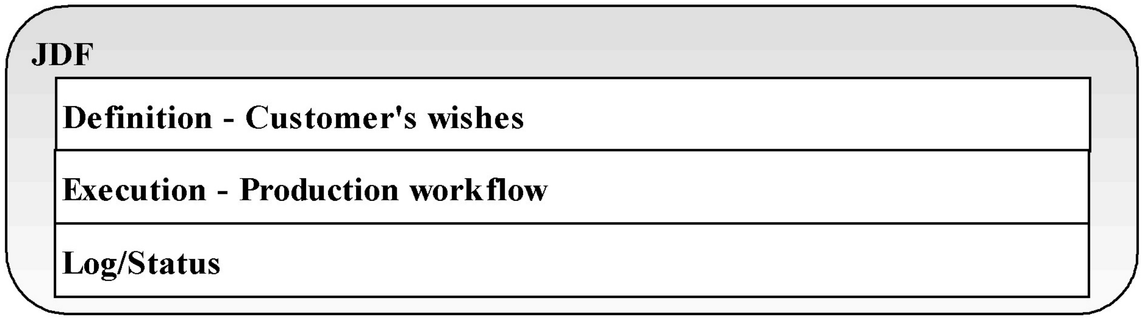

JDF
Definition - Customer's wishes
Execution - Production workflow
Log/Status

Figure 2.1 Overview of what JDF covers (inspired by [42])

2.2.1. Structure

A product defined in JDF in its entirety is called a *Job*. A Job is a tree of *Nodes* which describes different stages of a print job. A JDF Node can be of different types:

- **Product Intent Node** – A description of a print job which does not necessarily contain any actual processing information. It is rather a definition of a product from a customer's point of view, with possibly little knowledge about the processing steps. For example may a customer describe a brochure with 4 pages in color, but the actual processing steps are left out, such as RIPing, trapping etc.
- **Process Node** – This is a definition of a single processing step in a JDF workflow, for example the process of printing. Process Nodes are always leaves of a tree. The JDF Specification defines all the processes needed to complete a print job. Some example of process nodes are *Approval*, *BoxPacking*, *ColorCorrection*, *Rendering* which are briefly explained in Appendix B: Glossary.

To cluster processes that belong together or to enable multiple processing steps at the same time, there are also nodes of type **Process Group** and **Combined** respectively.

Process Nodes and *Resources* are the basic elements within JDF. Each Process Node is dependent on some input Resource/Resources that must be available in order for the Node to execute. When all the input Resources for a Process Node are available, execution may be performed and the processing is carried out and produces some output Resources/Resources (see **Error! Reference source not found.**) which in turn may be an input resource of another process. Resources are thus either consumed or produced by a Process. For example the Process of printing requires that ink, papers and plates etc. are available. The Nodes and Resources are defined as XML-constructs.

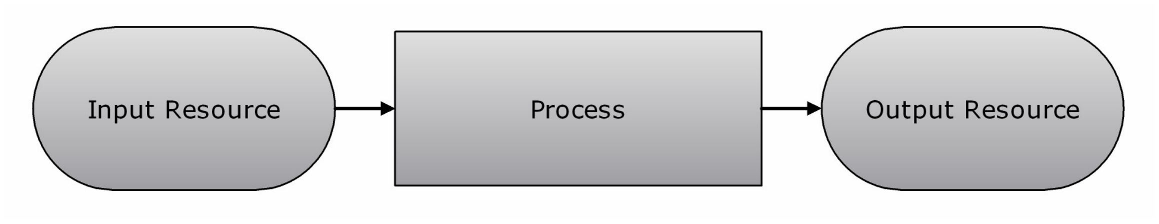


Figure 2.2 Resource consumption and production in JDF

2.2.2. Workflow

Print jobs in JDF are defined in such a way that the execution order of different parts of a job is not important. Most customers have a quite precise idea on how they want their product. What they may not know is the difference between choosing a regular four color scheme, black and white scheme or a more advanced color scheme in terms of cost and quality. The job described by Product Intent Nodes can vary from very general to extremely detailed, therefore the different knowledge of customers is not a problem.

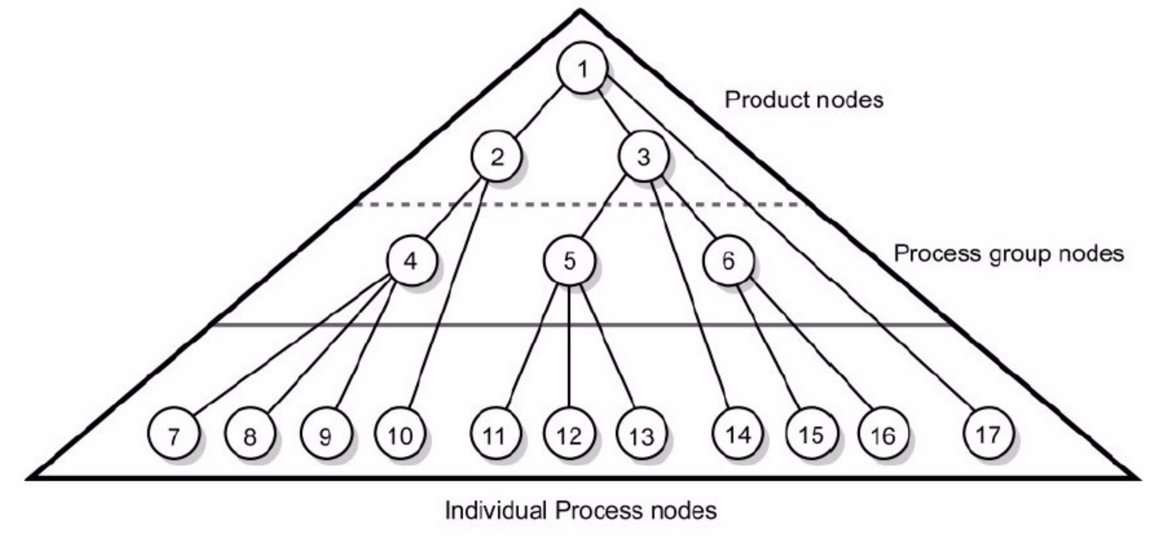


Figure 2.3 Tree of JDF nodes describing a brochure (image from [27])

Assume that Figure 2.3 is the tree of Nodes describing a brochure. Node 1 describes the entire brochure. Node 2 specifies the cover and the child nodes of 2 describe the work that needs to be done to complete the cover such as RIPing, plate making and printing. The same procedure is made with the contents of the brochure described by Node 3. It is up to the controlling system to plan and to overview the workflow, as well as deciding the order of execution. This production planning is usually made from a *Management Information System* (MIS, see below) which is the controlling system of a print production workflow. For example, the Process group which describes the process of printing the contents may be made before the printing of the cover, as long as the process of putting the cover and the contents together are made after the printing of both parts.

2.2.3. Roles

To avoid misconceptions and misuse of terminology the JDF Specification defines certain Workflow Component Roles [27] and their parts in a print production workflow:

- **Machine** - A machine is any part of the workflow system designed to execute a process. Most often, this term refers to a piece of physical equipment, such as a press or a binder, but it can also refer to the software components used to run a particular machine.
- **Device** - A Device is capable of executing the information it receives from an *Agent* or a *Controller*. Devices must be able to execute JDF Nodes and initiate its underlying machine. It may also be able to send messages using JMF back to the controller/agent. An example of a Device is the software that receives a Process Node and initiates for example a printing press to print what is defined in the Process Node.
- **Agent** - An agent can write, create and modify JDF Nodes. Anything that can be used to create a JDF Node is an agent, for example a simple text editor.
- **Controller -** A Controller can split up and route JDF Nodes to appropriate Devices. A controller must at least be able to initiate one process, controller or device.
- **MIS** - The Management Information System is controlling and overseeing the complete workflow and its units. It is also responsible for monitoring the progress of the job. The MIS can do this via JMF or by information from the JDF Nodes audits.

These roles are still quite vague in their definitions and most often a piece of software or even a machine can have many roles. It is also important to notice a hierarchy of Agents and Controllers is possible, where one controller can initiate another controller at a lower level. Throughout this paper there are also some more roles that are used and taken from the *Base ICS Specification* (see ICS Documents below) [5] which are defined in terms of interaction between different workflow components. These roles are:

- **Manager Interface** - The interface that sends JDF Instances and JMF messages to a Worker in a Device or a Controller.
- **Manager** - The software that implements the Manager Interface.
- **Worker Interface** - The interface that receives JDF Instances and JMF messages from a Manager in a Controller or a MIS.
- **Worker** - the software that implements the Worker Interface.

All these roles are used in this thesis and they are important in order to understand the meaning of a term. An instance of Elk (see The Elk Reference Implementation, p. 21) is a Worker that implements the Worker Interface (in terms of interaction with a Manager). It can also be defined as “a Device with Agent properties”. Alces (see Alces, p. 39) is a Manager and the figure below shows the visual relationship between the roles.

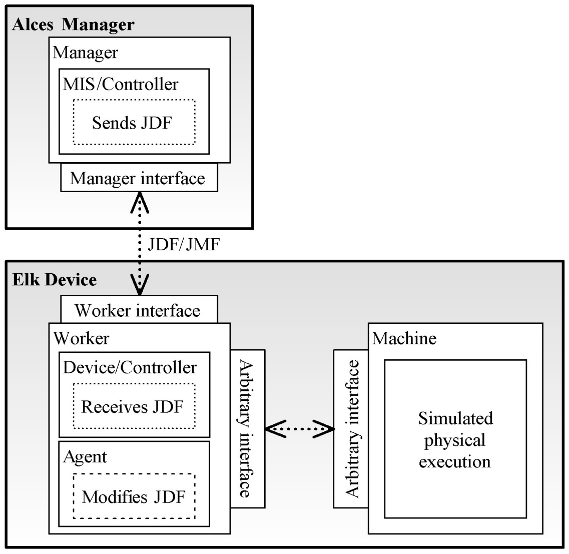


Figure 2.4 Elk and Alces roles

2.2.4. JMF

The Job Messaging Format (JMF) defines a unified communication channel for systems within a print production workflow. The need for interaction in a dynamic and standardized way is crucial in order to monitor and control the workflow. JMF provides a wide range of capabilities to accomplish these interactions. Some of the use cases when JMF is used are at system startup, for error tracking purposes, job changes, queue handling and job submission. A Device or Worker may update status changes to the MIS via JMF. JMF messages are most often sent over HTTP.

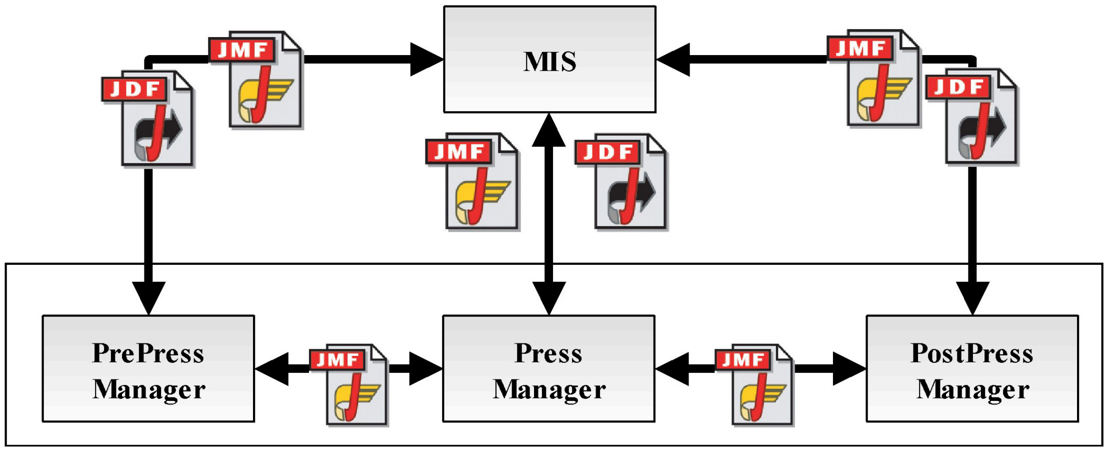


Figure 2.5 Interaction between MIS and a print production workflow

The MIS communicates with the different managers (Figure 2.5) or controllers which in turn communicate with underlying devices and controllers.

JMF Messages are divided into different message families to indicate their purpose. The different families are

- **Queries** which are sent to a Device and asks for its current state.
- **Commands** which are intended to modify the Device's current state.
- **Responses** which are answers to Queries and Commands.
- **Acknowledges** which in case the processing of a Query is time consuming is sent to confirm that a Query message was received.
- **Signals** which are sent from a Device or Controller to inform a MIS about a state change without the MIS explicitly asking for it.

2.2.4.1. Subscription Management

For a MIS to be able to monitor the workflow it needs to keep track of the Devices currently running and their progress. Therefore, a `Subscription` element can be included in a Query message and the receiving Device will then send back status information in a given time interval or at a given rate to a specified location (commonly a MIS or Controller URL). The location and the intervals are all submitted in the Subscription.

2.2.5. Queue Management

Many Devices, such as printers or software will be able to receive jobs even though they are currently processing another job. The Device has an internal queue where jobs are stored and executed as the jobs are being processed. A MIS may be interested in the status of the queue or current jobs in the queue. JMF also defines and supports queue management.

2.3. ICS Documents

The JDF Specification is very large document because it covers all the processes needed to complete a print job. A product or a piece of software that claims to be JDF enabled does not likely implement the whole of JDF. Rather, it implements a small subset of JDF which corresponds to the tasks it can perform. It is up to each vendor to implement any subset of JDF. There is nothing which guaranties that two products can interact even though they both claim to be JDF compatible.

The solution to this interaction problem is a set of *Interoperability Conformance Specification* (ICS) documents [28]. These documents were introduced together with version 1.2 of JDF to avoid incompatible JDF "dialects" and to make systems interact in a meaningful way. At the time of writing there are 6 categories of ICS documents and they each define a subset of JDF. The different categories focus on different product groups such as Prepress, ConventionalPrinting, and Binding. There are also the Base ICS document which defines the minimal requirement for a Worker and the *MIS ICS* which defines the communication requirements between a MIS and production equipment. Figure 2.6 shows an overview of the ICS documents.

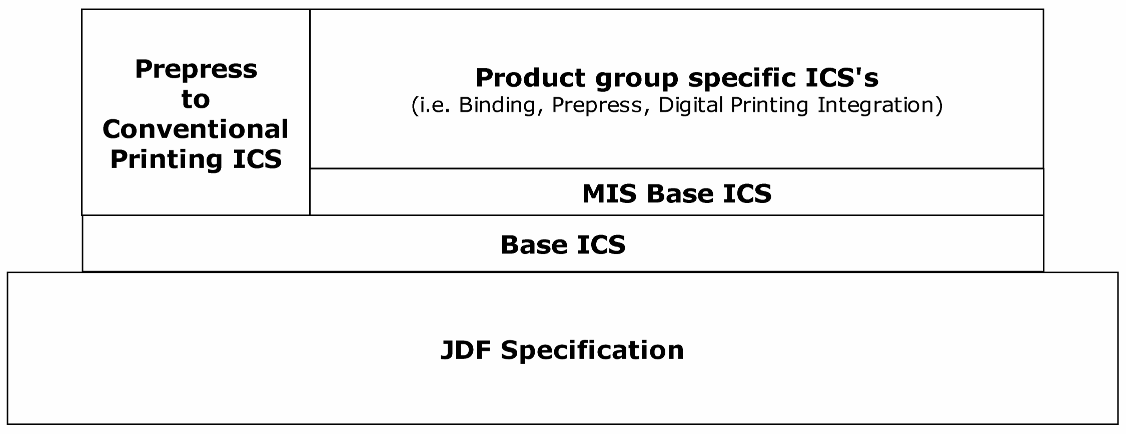

Prepress to Conventional Printing ICS
Product group specific ICS's
(i.e. Binding, Prepress, Digital Printing Integration)
MIS Base ICS
Base ICS
JDF Specification

Figure 2.6 Overview of the structure of the ICS Documents

To additionally increase the flexibility each ICS document defines different levels of interoperability.

3. The Elk Framework

This thesis describes the process of developing a JDF enabled software tool; the programming tasks that needs to be resolved; and the generic functional requirements for such a tool. The Elk Framework [13] is the resulting software and consists of:

- An Application Programming Interface (API) and
- A reference implementation of the API

As Buckwalter suggests in the article "*JDF-enabled Workflow Simulation Tool"* [8] certain services are needed in order to fulfill the requirements of a Device implementing the *Worker Interface* (see section 2.2.3 Roles p. 13). The API is built upon the specification of the services summarized below and the API also includes some other features which alleviate implementing a Device in good programming manners. Below is a figure of the high level components of the Elk Framework.

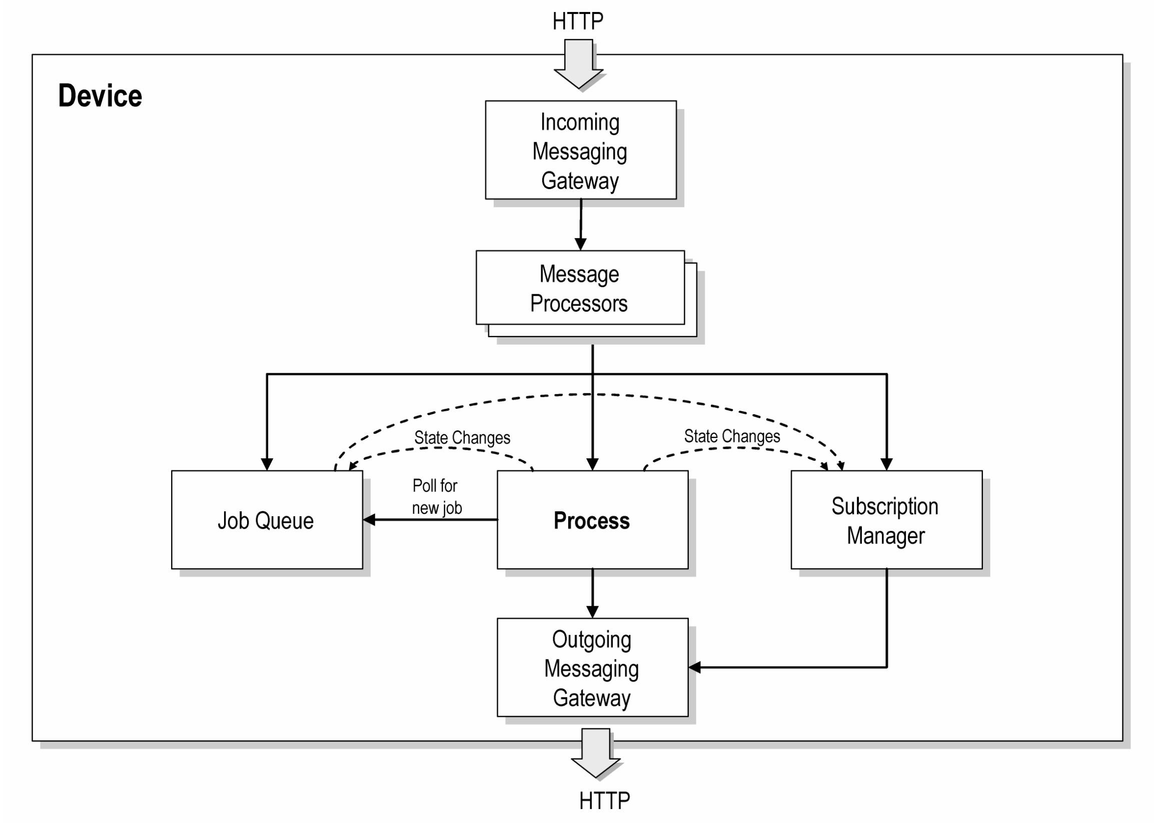


Figure 3.1 Overview of the Elk Framework (image from [8])

3.1. Message Gateways

Communication is an essential part of JDF software. The interaction between a Worker and the MIS is done through JMF and enables the Worker to update the MIS on state changes, amount of printed sheets for a printing press, the Device's queue and running jobs etc. The Elk Framework provides message gateways in order to receive JDF and JMF instances without exposing the underlying network protocol. This will enable the user to focus on the processing of messages instead of *how* the JDF and JMF messages are sent and received. The processing is done through the *Message Processors*. The Elk Framework also contains an outgoing message gateway which adds the necessary http headers and so forth to the outgoing JMF messages.

3.2. Message Processors

The Message Gateway routes the JMF messages to a Message Processor. A Message Processor is designed to handle a JMF message [27] of a specific type, for example a QueueStatus message. The Elk Framework contains an interface (`JMFProcessor`) that defines a generic method for processing a JMF message. This enables any implementing classes to be defined and handled in a unified way.

3.3. Subscription manager

The subscription manager is the piece of software that handles the subscriptions and persistent channels. It is responsible for broadcasting events to subscribers. It also stores and keeps track of subscribers. The subscription manager removes some work from the implementing process. Handling of subscriptions can be implemented separately which simplifies the implementation of the process. Furthermore, the subscription handling is similar independently from which process is implemented. The `SubscriptionManager` interface specifies this functionality.

3.4. Job queue

Submitted jobs to the Device are stored in a Job queue, thus the functionality of a queue is necessary. The Queue management in JDF is quite detailed. The Elk Framework provides an interface defining the generic functionality of a queue; `Queue`.

3.5. Process

As mentioned, a process is the machine or device that consumes input resources and produces some kind of output resource. The Process is performing the actual work. The `Process` interface defines a set of generic methods that all processes need to handle. Some examples of methods that are defined here are starting, initiating, stopping and running the device. A vendor that wishes to use Elk as a foundation for implementing JDF enabled software in their company or print shop needs to implement the `Process` interface. As the Elk Reference Implementation (ERI, see below) is constantly enhanced and more functionality is added to it, much of the messaging and other requirements are already implemented and can be reused. However, the Elk does not contain any machine or device specific details on how communication shall be carried out with the actual machine.

The above interfaces are the basic JDF specific services discovered and implemented in the Elk Framework. The Elk Framework also provides some additional functionality classes and interfaces which are briefly explained below. Most of these classes alleviates the programming and implements some basic functions which are useful in implementing any kind of JDF Software.

3.6. Factory

The Elk Framework provides a `JDFElementFactory` to easily create JDF Elements. The Factory is a Singleton [50], meaning that only one instance of the object exists and is used to create JDF Elements throughout the Elk application. The Elk is using and is dependent on the CIP4 developed JDFLib-J [30] and it is due to changes and has a different development cycle than Elk, new releases does not coincide with Elk releases. Therefore compatibility issues may arise when updating or changing to a new version of JDFLib-J. One of the advantages using the factory is that if the implementation of creating JDF Elements change, the only place where any code needs to be changed is in the implementing method of the Factory. The Elk Framework offers to create JDFElements in different flavors and to encapsulate the concrete implementations of how the elements are created, adhering to the Abstract factory pattern [1]. The client (e.g The Elk Reference Device) only need to worry about the `JDFElementFactory` method `createJDFElement(String elementName)` in order to get the Element, nothing else (see figure below).

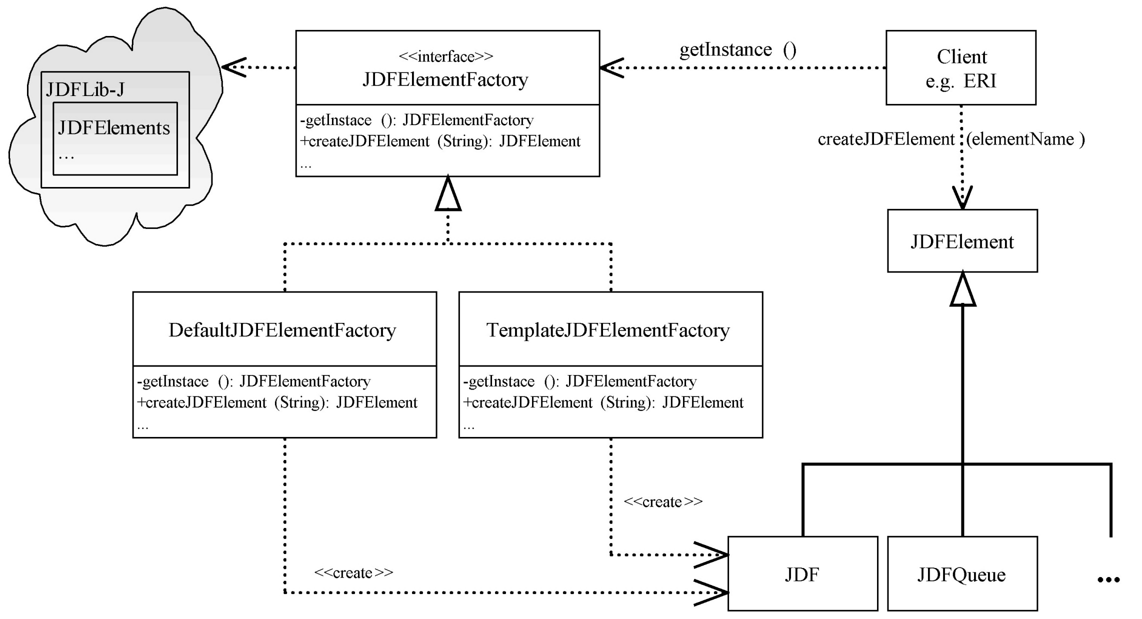

JDFLib-J
JDFElements
...
<<interface>>
JDFElementFactory
-getInstace (): JDFElementFactory
+createJDFElement (String): JDFElement
...
getInstance ()
Client
e.g. ERI
createJDFElement (elementName)
JDFElement
DefaultJDFElementFactory
-getInstace (): JDFElementFactory
+createJDFElement (String): JDFElement
...
TemplateJDFElementFactory
-getInstace (): JDFElementFactory
+createJDFElement (String): JDFElement
...
<<create>>
<<create>>
JDF
JDFQueue
...

Figure 3.2 Abstract Factory pattern in Elk, UML

Which `JDFElementFactory` to be instantiated can be made in different ways and in Elk it is defined in a Java properties file.

3.7. Events and EventListeners

In order to achieve as much decoupling (see 5.5 Decoupling) between objects as possible the Elk Framework defines certain Event classes for different type of events that may occur in a Device. All events extend the `ElkEvent` interface. Events are used in Elk to internally communicate state changes. These internal events may be translated into external events in the form of Signals (see JMF, p. 14). Signals are used to inform the MIS or a Controller about the status of the Device, the Queue or the progress of a job. Classes for events which are defined in the Elk Framework are:

- **ProcessStatusEvent** – This type of Events are supposed to be generated on status changes for the Device, for example from Idle to Running or from Running to Cleanup.
- **ProcessAmountEvent** – ProcessAmountEvents are generated during the execution of a job when the process produces an amount of some sort. Most commonly the Amount has to do with the amount of printed sheets.
- **ProcessQueueEntryEvent** – Each QueueEntry in a Queue has a Status depending on how far in the execution process the QueueEntry and its attached JDF Node has come. When a QueueEntry is in the Queue waiting to be executed, the status is "Waiting", once the execution start it is changed to "Running". The resulting status of a QueueEntry is either "Completed" or "Aborted". The ProcessQueueEntryEvents are currently used to update the Queue on changes of status in a QueueEntry.
- **QueueStatusEvent** – This type of events are used when the Queue change its state.

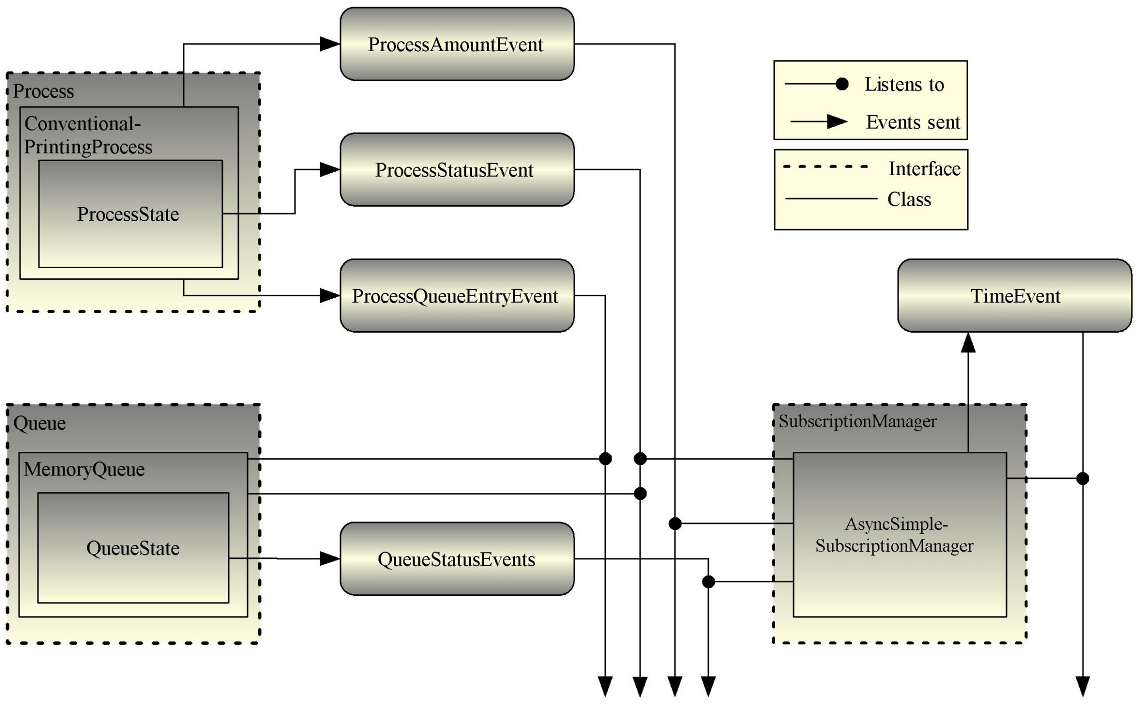


Figure 3.3 Overview of the Elk Reference Implementation event handling

Note: The image above is a simplification, the Events are sent using helper classes, Notifiers which are class variables of the process. The reason for this can be found in section 5.5.1 Observer pattern on p. 29.

3.8. Filters

The Elk `DeviceFilter` interface defines a method for filtering a `JDFDeviceList` according to a given `JDFDeviceFilter`. The `QueueFilter` interface defines a method for filtering a queue according to a given `QueueFilter`. The Elk Framework provides these interfaces to decouple the behavior (filtering, sorting) of a `JDFDeviceList` or `JDFQueue` from the object itself. This design solution is the Strategy pattern (see Strategy pattern, p. 32).

3.9. Configuration of the Device

In JDF, a device and its capabilities are defined as an XML-element with sub-elements and attributes which reveals what the Device is capable of doing. The Elk Framework provides specifications for accessing and configuring the Device via the `Config` and `DeviceConfig` interfaces. These definitions are used throughout the application to retrieve device specific information such as its id, which protocols it has support for when submitting a JDF and so on.

3.10. Summary of the Elk Framework

To summarize, the Elk Framework API acts as a skeleton upon which a JDF Device can be built. It provides some generic functionality which alleviates the implementation obstacles that needs to be solved in order to deploy a functional JDF enabled device. It mainly consists of interfaces, the interaction glue between different parts of a device. The Elk Reference Implementation will put some meat on this skeleton and act as an example application and implement the interfaces specified in the Elk Framework.

4. The Elk Reference Implementation

The architecture and design of the ERI is most easily explained by an image, see Figure 4.1.

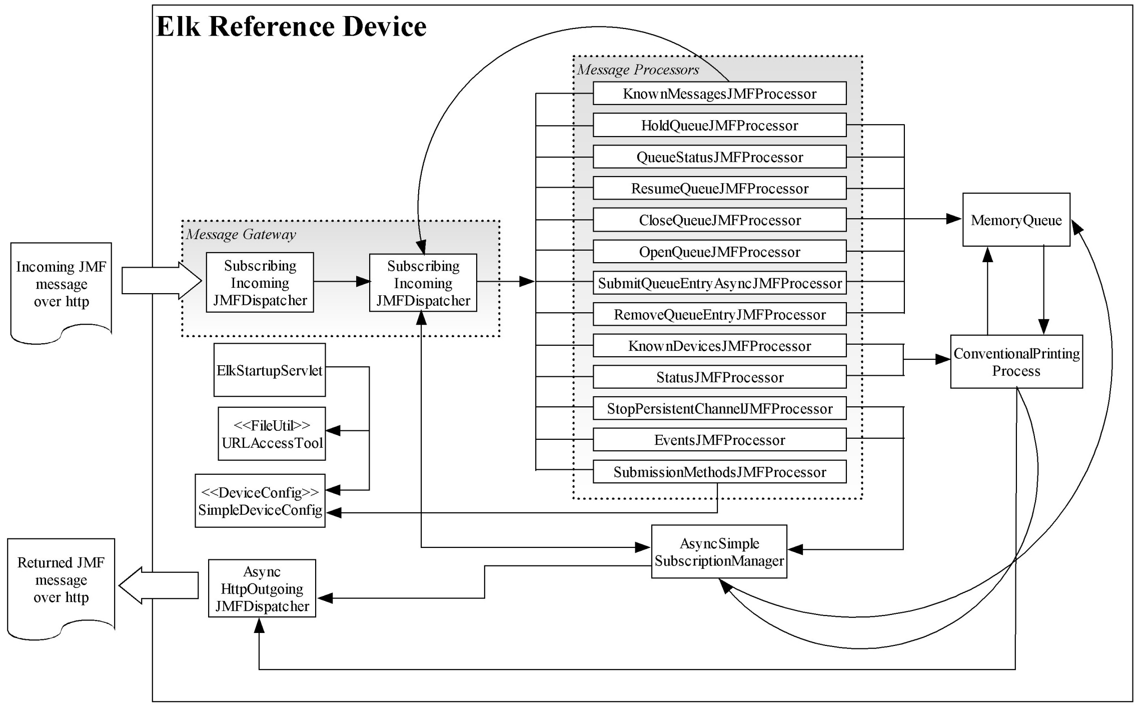

Elk Reference Device
Message Processors
KnownMessagesJMFProcessor
HoldQueueJMFProcessor
QueueStatusJMFProcessor
ResumeQueueJMFProcessor
CloseQueueJMFProcessor
OpenQueueJMFProcessor
SubmitQueueEntryAsyncJMFProcessor
RemoveQueueEntryJMFProcessor
KnownDevicesJMFProcessor
StatusJMFProcessor
StopPersistentChannelJMFProcessor
EventsJMFProcessor
SubmissionMethodsJMFProcessor
MemoryQueue
ConventionalPrinting Process
Message Gateway
Subscribing Incoming JMFDispatcher
Subscribing Incoming JMFDispatcher
Incoming JMF message over http
ElkStartupServlet
<<FileUtil>> URLAccessTool
<<DeviceConfig>> SimpleDeviceConfig
AsyncSimple SubscriptionManager
Returned JMF message over http
Async HttpOutgoing JMFDispatcher

Figure 4.1 Conventional Printing Elk Device

The ERI implements the services defined in the Elk Framework. The classes being used in an instance of Elk are configured through an xml file managed by the Spring Framework [53]. Since Spring uses dependency injection (see 9.2 Dependency injection, p. 42) to instantiate its objects, each instance of Elk can easily be configured, and new classes can be replaced by older ones in a convenient way. Below is a quick explanation of the classes in the figure above:

4.1.1. ElkStartupServlet

This servlet's sole purpose is to configure the Elk reference implementation. It loads the Spring [53] configuration that instantiates objects and injects dependencies. It also loads relevant event listeners to the classes that listen to events.

4.1.2. URLAccessTool

A file utility class used to access the file system. The `URLAccessTool` implements the `FileUtil` interface. The files are addressed using URLs.

4.1.3. SimpleDeviceConfig

The ERI provides a `SimpleDeviceConfig` class which supports loading the configuration from an XML-file on the file system or on the network through a URL. Implementing the `DeviceConfig` interface (see Configuration of the Device) will enable the user to build another, more sophisticated way of configuring the Device.

4.1.4. DispatchingJMFServlet and SubscribingIncomingJMFDispatcher

The `DispatchingJMFServlet` receives, parses and forwards JMF messages to the `SubscribingIncomingJMFDispatcher` (`JMFDispatcher`). The `JMFDispatcher` then identifies the type of the message and forwards it to the correct JMF message processor.

4.1.5. AsyncHttpOutgoingJMFDispatcher

`AsyncHttpOutgoingJMFDispatcher` is the class that asynchronously dispatches JMF Messages which most often is a Response or a Signal to a receiving MIS or another Controller/Device.

4.1.6. Implemented Message Processors

The Elk Reference Implementation supports the following messages:

- **Queries**: Events, KnownDevices, KnownMessages, SubmissionMethods, QueueStatus
- **Commands**: StopPersistentChannel, CloseQueue, HoldQueue, OpenQueue, RemoveQueueEntry, ResumeQueueEntry, SubmitQueueEntry, RemoveQueueEntry

The corresponding implementing classes follows the naming convention to include the message name followed by `JMFProcessor` e.g. `KnownDevicesJMFProcessor` for the KnownDevices message. Since the Elk is under constant development, the implemented messages changes over time.

4.1.7. AsyncSimpleSubscriptionManager

The current version of Elk uses the `AsyncSimpleSubscriptionManager` to handle the subscriptions. The `AsyncSimpleSubscriptionManager` listens to `ProcessStatusEvents`, `ProcessAmountEvents` and `QueueStatusEvents` in order to be able to broadcast Signals and Status messages to the subscribers. See Figure 3.3 Overview of the Elk Reference Implementation event handling.

4.1.8. MemoryQueue

The ERI implements a memory based queue. That is, the entries in the queue are lost each time the Elk Device restarts or the server shut down. The `MemoryQueue` implements the `Queue` interface. Of course a persistent queue would be desirable (see 6.2 Future work, p. 38).

4.1.9. ConventionalPrintingProcess

See Implemented Processes below.

4.2. Implemented Processes

At the moment the ERI contains of two implemented simulated processes defined in JDF.

4.2.1. ApprovalProcess

The Approval process is usually carried out by a person who will approve or disapprove for example a sample print. The current implementation simulates this by approving half of the jobs and disapproving the rest. It is a very simple and first proof that the ERI can be used for simulating processes. The process accepts JDF Jobs containing the Process Node of `Type` “Approval”.

4.2.2. ConventionalPrintingProcess

The implementation simulates a printing press. This process executes JDF Jobs containing Process Nodes of `Type` “ConventionalPrinting”. The process simulates printing of sheets at a certain speed and events that can be subscribed are generated for each of the printed sheets. The simulation also produces a certain amount of waste. All of these properties are configurable.

The implemented processes inherit and implement the functionality of the Elk Framework's `Process` interface. Many of the inherited methods and much of the functionality overlap, therefore two classes that do the work of implementing common process behavior are included in the ERI. Both of these could preferably be reused when extending Elk to handle another Process Node `Type`. Figure 4.2 below shows the class hierarchy.

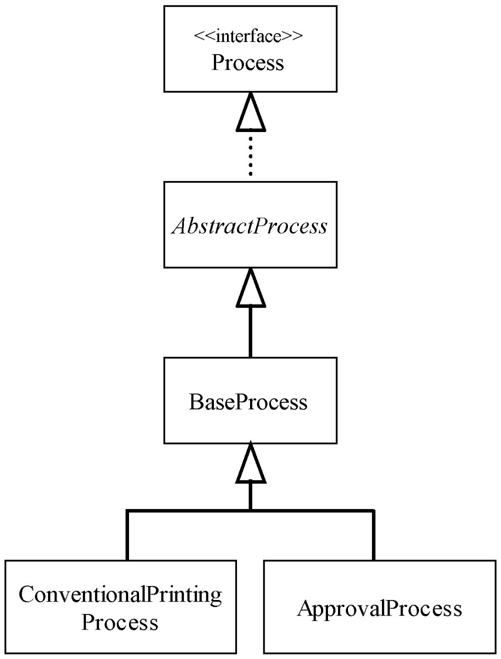


Figure 4.2 The Process implementation hierarchy, UML

4.2.3. AbstractProcess

Since certain classes are interested in the status of a process, methods for registering and deregistering listeners to process events are needed. These methods are common to all processes and are implemented in the `AbstractProcess`.

4.2.4. BaseProcess

Functionality common to all processes also includes:

- The process of fetching ready to run jobs from the queue.
- Selecting only the Process Nodes of correct `Type` and forward it to the executing implemented process.
- To ensure that the JDF Node is ready to execute and that the Input resources are available.
- To add and modify audits for executed JDF Nodes.
- Firing of events on status changes in the Device.
- To perform post processing of a job including cancel affected subscription and sending of the processed JDF/JMF to its correct location.

Some of other basic functionality includes starting and stopping of the device which is also implemented in the `BaseProcess`.

5. Program Design, Concepts and Development

The Elk Framework strives to use good programming practices and demands to be robust and reliable is very important for a framework. In order to accomplish this, the project evolves through an iterative development process [22] where redesign and changes are parts of development. Firstly, in this paragraph a brief explanation of the concepts are explained, the following sections go into detail into each concept and design choice. To have as much control of the execution sequence of Elk as well as for debugging purposes, Logging is a major feature. Testing is important for any programming task and for a framework such as Elk where implementations change and evolves it is even more important. The ease to extend, to develop Elk further is of course also essential, one example of this is given in the section Extensibility. Internal changes and reuse of code without changing the external behavior, changing the API is called Refactoring and is one of the programming concepts that are used. Different methods to accomplish Decoupling, to have as little dependency between objects have also been a goal. All these practices together with well documented code have resulted in maintainable code, see section Maintainability.

All of the above practices hold for more or less any programming project, the use of Asynchronous processing and Preprocessing of received JDF files are of a more JDF/Elk specific nature. They are further explained in the just mentioned sections.

5.1. Logging

"Logging provides a way to capture information about the operation of an application. Once captured, the information can be used for many purposes, but is particularly useful for debugging, troubleshooting, and auditing." [40].

The main disadvantage of using logging as a debugging tool is the fact that it adds extra code into the application which leads to code overhead and loss of performance. However, the code overhead is of little disturbance and can actually be a source of information for new developers trying to understand the code plus the ability to follow the control flow of the application in the outputted logs. The Elk uses log4j [39] as its Logging framework which has been implemented with an emphasis on speed [40].

The Elk Framework is used and developed by several authors; therefore the information in each log message is maximized. The logging strives to inform the reader, the user or the developer of the current state of the application and its fields. This is also applicable for the messages of exceptions in Elk, which most often is also sent to the log. The principle "To capture the failure, the string representation of an exception should contain the values of all parameters and fields that 'contributed to the exception'" [7] is used in Elk and also applied for other type of messages.

Two other features that advocates logging before regular debugging tools are:

- When debugging an application, the values registered and the state of the application is transient while the debug log messages are stable and available for later analysis and evaluation [36].
- Most debugging tools have difficulties handling applications that run in different threads which is the case for Elk. Therefore, the logging from different threads can either be sent to different output files or the messages are simply logged in the order they occur.

Important logging features that Log4j [39] provides is dynamic configuration, logging levels, directed output, and enabling/disabling logging. Log4j is an open source framework which is easy to use and to configure. The configuration can be made in different ways and Elk configures the properties of log4j through an XML-file in the web application directory (WEB-INF/classes). It is possible to control the amount of log messages by setting the debug level. The debug levels have priorities where FATAL has highest priority followed by ERROR, WARN, DEBUG, and INFO. Another concept of log4j is the Appenders. Appenders are used to direct the log messages to different sources such as console, files, GUI components, remote socket servers and more. The Elk uses a so called RollingFile appender where log messages are sent to a file which is replaced by a new one when it has reached a specified size. When debugging is disabled the loss of speed is extremely little and not of importance to overall performance.

5.2. Testing

Testing is "the process used to help identify the correctness, completeness, security and quality of developed computer software"[52]. Testing software can me made in different ways and with different aims:

- **White box testing** – The source code of the application or the units under tests are available to the tester and can be accessed freely. This method of testing is often useful when testing smaller units of code (such as classes in java). These unit tests often compare an expected result with the actual result, errors can be found at an early stage in development. The developers do these tests themselves and may not cover all scenarios because of home blindness.
- **Black box testing** – The application is tested without knowledge of the source code. The functionality of a system is tested and is similar to user behavior. One form of black box testing is the beta releases of software where users can try out new features and report bugs to the vendor.

With big projects and framework software development continuous testing is essential since refactoring (see Refactoring, p. 28) and additions are common. A change or an addition to code can break something which is not directly connected to the piece of code that is modified. The process of running the same tests over and over again is called regression testing [52].

There are a few obstacles to overcome in order to make testing useful and used. Firstly, it needs to be easy to run, preferably with a single command. Secondly, the tests need to be extensive so that code can be changed; tests rerun and for sure catch any breaking of code. The principles of writing tests before the actual code is written and also to run regression tests are adhered in Extreme Programming [18][17] and also a goal for the Elk development team.

One of the more widely accepted testing procedures is to use the JUnit [34] framework which enables the user to “cheaply and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus on your development efforts.” [35]. Junit is a very light weight framework which has two primary classes, the `TestCase` class and the `TestSuite` class. A typical scenario is to write a TestCase for each of the project’s classes, put them together in a `TestSuite` and then run the test suite at a regular basis. When classes are added to the project, new `TestCase`s are added to the `TestSuite` and are thus included in the regression tests. Refactoring can be done “much more aggressively once you have the tests” [35] because in case something breaks it will show up in the tests.

5.2.1. Testing in Elk

The Elk regularly runs two types of tests: Unit tests using JUnit (White box testing) and Black box testing using the external tool Alces [2].

- The JUnit tests are run using the `test` task in Ant (see Apache Ant, p. 43) which runs all the tests for Elk. When new source code is committed to CVS (see Maintainability, p. 33) the tests are automatically run using the tool CruiseControl [10]. With this feature regression tests are carried out automatically. CruiseControl enables developers to be notified via email about compilation and test results.
- Alces (see Related work, p. 38) automatically sends a set of JMF-messages to Elk and summarizes the test results in a XML test report. Initiation of testing Elk’s functionality towards a Manager (see Roles, p. 13) does still have to be carried out manually even though the actual tests are automatic once they are initiated.

5.3. Extensibility

The Elk Framework consists of interfaces and abstract classes which attempts to define *what* methods are needed, not *how* they should be implemented or in how much detail. It is up to the implementing classes how the methods perform their tasks as long as it coincides with the interface. In case a particular user is interested in a very specialized method of filtering a queue the Elk Framework API has an interface `QueueFilter`. The interface defines one single method `filterQueue` which have two arguments: the Queue to be filtered and the `JDFQueueFilter` to apply to the Queue. The Elk Reference Implementation provides a few classes for filtering Queues:

- **SortingQueueFilter** which sorts the Queue according to the JDF Specification’s sorting of QueueEntries in a Queue. This filter ignores the JDFQueueFilter argument.
- **AttributeQueueFilter** which sort the Queue and filters it according to some predefined attributes. The QueueFilter argument is ignored.

- **BaseICSQueueFilter** which filters the Queue according to the given `QueueFilter`. The class handles only elements and attributes of the JDFQueueFilter that are required to conform to the BaseICS [5] conformance specification.

A possible extension to these classes would be to have an implementation of a `QueueFilter` which handles all attributes of the given `JDFQueueFilter`. A new implementation could easily be implemented and used in an Elk Device without having to worry about compatibility, as long as the new class implements the `QueueFilter` interface. Also see section "Strategy pattern" and "Figure 5.6 QueueFilter and the Strategy pattern, UML" on page 32.

The above example shows how an extension can be made to enhance Elks functionality. The same extensibility options apply to all basic constructs needed to implement a fully functional JDF Device.

5.4. Refactoring

The term refactoring [48] implies that a programming project that is constantly growing is also amendable for redesign and enhancements under its life cycle. The ability to break out common parts, to program to interfaces and to avoid duplication of code is a widely accepted programming practice. As new features are built in to the project, other already implemented features may be reused and extracted. One particular case for the ERI is the Subscription management. At an early stage the Subscribing management was all implemented in a single class but as features were added the class grew out of hand. The solution was to refactor the class into several new ones, each responsible for a specific task of the subscribing mechanism. The refactoring resulted in a new subscription package which included one class for registering subscriptions, one class for deregistering subscriptions, one for storing subscription etc. The class diagram (Figure below) shows an overview of the subscription package as it designed in the current implementation (some classes left out).

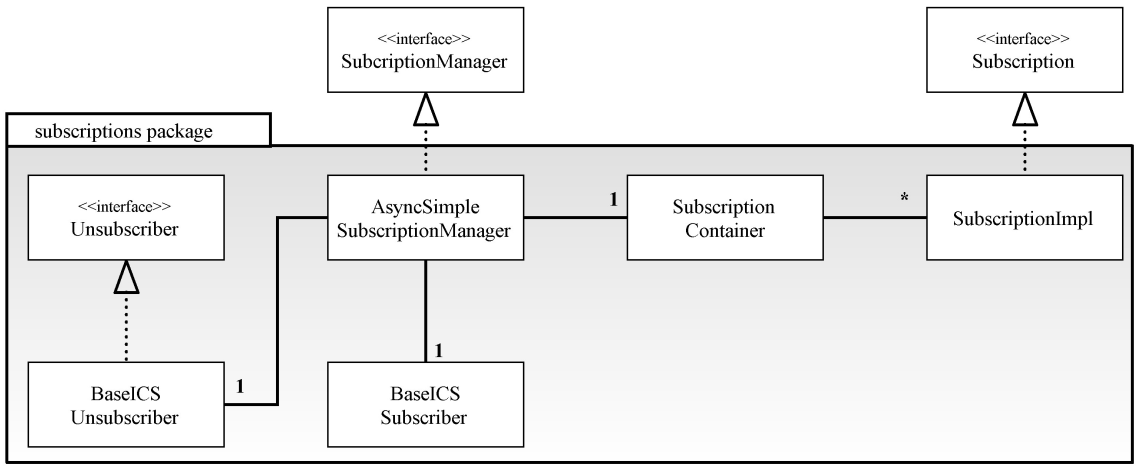


Figure 5.1 Overview of the subscriptions package, UML

The next step in the development of the Subscription package is to enable persistent storage of the subscriptions on disk (see 6.2 Future work, p. 38). The current implementation loose any subscriptions on application shut down. A suggestion for the next refactoring step would be to extract an interface from the `SubscriptionContainer` and implement a new `PersistentSubscriptionContainer` class. The former `SubscriptionContainer` can be renamed and reused in case memory-based subscriptions are desired in the application (Figure below).

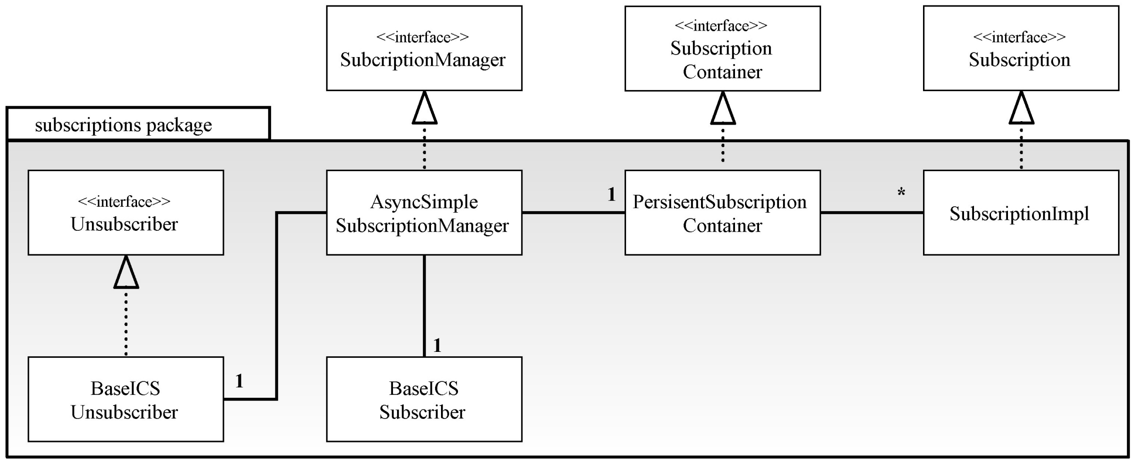


Figure 5.2 Overview of refactored subscriptions package, UML

5.5. Decoupling

The JDF Specification [27] defines a very large amount of Processes and Resources. JDFLib-J [30] provides methods for accessing thousands of attributes and elements in a JDF file, many of them auto-generated. The Elk Framework implements only a subset of these methods and has the focus on the Base ICS [5] specification. Despite that, the Elk enables great extensibility options (see Extensibility, p. 27), partly because of the aim to restrict dependencies between objects as much as possible. The practice to make objects independent from each other is called *decoupling*. One advantage which decoupling conveys is that an implemented class (an object) can easily be replaced by another. The idea of programming to interfaces is the foundation to achieve decoupled software. The Elk is using different strategies, design patterns to solve different particular problems. The design patterns that are used look very similar in structure but solve different programming problems. The relationship between the patterns explained below is discussed at the end of this section.

5.5.1. Observer pattern

One way to accomplish decoupling is by using event messaging and the Observer pattern. The Observer pattern is “used in computer programming to observe the state of an object in a program” [45].

The aim is to decouple the Sender/Invoker from the Receiver/Listener so that they can act independently from each other. Example:

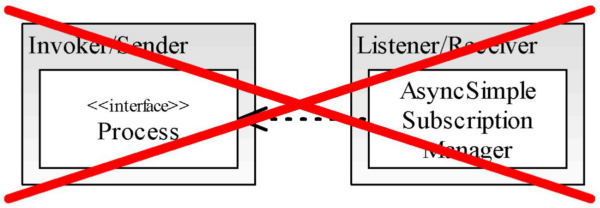


Figure 5.3 Strongly coupled Sender and Receiver classes

Above the `AsyncSimpleSubscrptionManager` is dependent on the `Process` interface. The example assumes that the `AsyncSimpleSubscriptionManager` is interested in the Process status changes in order to distribute it to its subscribers. The strong coupling between these objects is not good. Replacement of one class would require changes in both them. One solution to the problem is to use the Observer pattern and extract an Observer/Listener interface. The Elk also uses an extra class, a *mediator*, `ProcessStatusListenerNotifier` to notify the registered Listeners. The Mediator pattern [38] makes an extension of the Observer pattern and instead of having the subject/sender itself being responsible of the notification to the observers it is the mediator. The resulting class diagram would look like in the figure below.

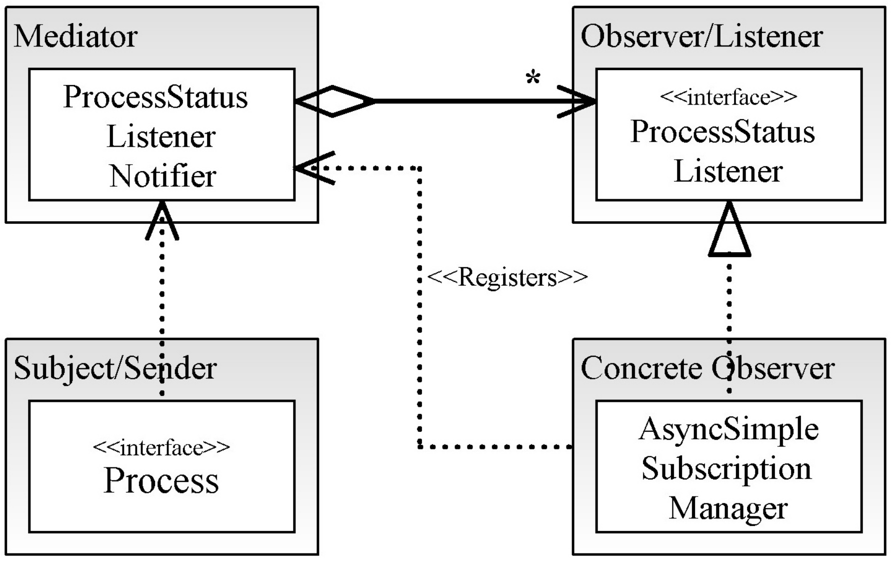


Figure 5.4 Decoupling using the Observer/Mediator pattern, UML

The `ConventionalPrintingProcess` which implements the `Process` interface acts as the subject/sender or the class that generates/fires an event (see also Figure 3.3 Overview of the Elk Reference Implementation event handling). The `AsyncSimpleSubscriptionManager` (`SubManager`) is a listener which can perform actions independently of the `Process`. During the running of the application when receiving an event; it does not need to know about the functionality or the implementation of the `Process`, neither does the `Process` need to know anything about the implementation of the `SubManager`. Upon application startup the application needs to register the `SubManager` to the `ProcessStatusListenerNotifier`. In case another class needs to listen to `ProcessStatusEvents`, the only thing that needs to be done is to register another listener which implements a different behavior upon the occurrence of an event (e.g. which is the case for the `Queue`).

Summary: The Elk uses the observer/mediator pattern to accomplish decoupling and it is used to listen for state changes in the device.

5.5.2. Command Pattern

Command pattern is a design pattern in which objects are used to represent actions. A command object encapsulates an action and its parameters [9]. In Elk this is used for the `JMFProcessor`s. The incoming `SubscribingIncomingJMFDispatcher` (`JMFDispather`) only knows about the `JMFProcessor` interface and the action to `processJMF`. The implementation of each `JMFProcessor` and the actions to be taken depending on the JMF message received is chosen at runtime. A JMF message is received and dispatched to the appropriate processor without knowledge of the actions being carried out. See Figure 5.5, below.

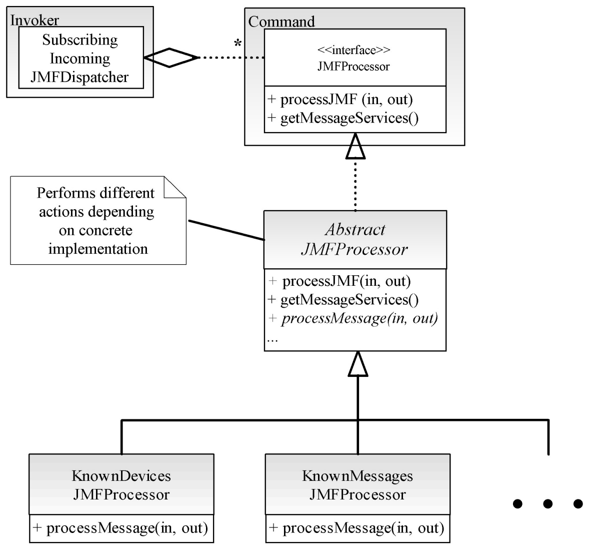

Invoker
Subscribing Incoming JMFDispatcher
*
Command
<<interface>> JMFProcessor
+ processJMF (in, out)
+ getMessageServices()
Performs different actions depending on concrete implementation
Abstract JMFProcessor
+ processJMF(in, out)
+ getMessageServices()
+ processMessage(in, out)
...
KnownDevices JMFProcessor
+ processMessage(in, out)
KnownMessages JMFProcessor
+ processMessage(in, out)
• • •

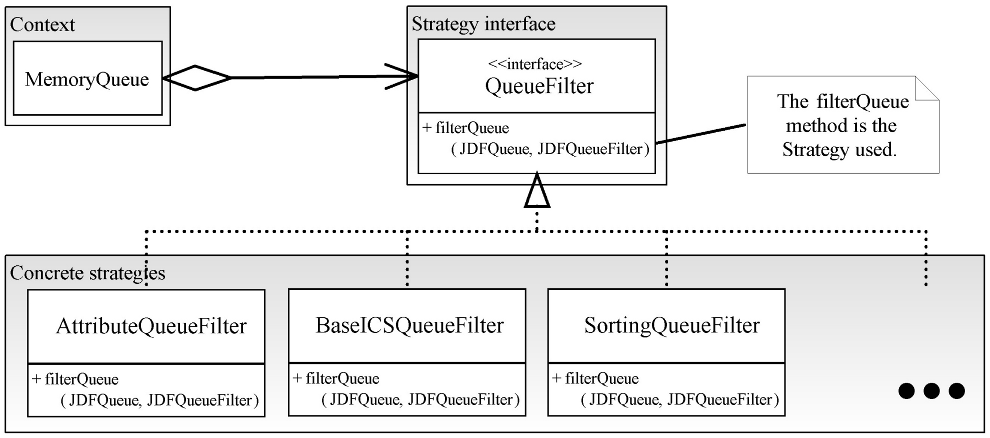
Figure 5.5 Message processors and the Command pattern, UML

In the figure it can also be noted that Elk uses the `AbstractJMFProcessor` which implements the common message behavior and the concrete command objects (`KnownDevicesJMFProcessor`, `KnownMessagesJMFProcess` and so on) implements the message specific actions.

Summary: The Elk uses the Command pattern to accomplish independency between the `JMFDispatcher` and the actions to be taken in JMF message processing.

5.5.3. Strategy pattern

The "Strategy pattern is a particular software design pattern, whereby algorithms can be selected on-the-fly at runtime." [55]. One example of the use of the Strategy pattern is in the `MemoryQueue` class.



Context
MemoryQueue
Strategy interface
<<interface>>
QueueFilter
+ filterQueue (JDFQueue, JDFQueueFilter)
The filterQueue method is the Strategy used.
Concrete strategies
AttributeQueueFilter
+ filterQueue (JDFQueue, JDFQueueFilter)
BaseICSQueueFilter
+ filterQueue (JDFQueue, JDFQueueFilter)
SortingQueueFilter
+ filterQueue (JDFQueue, JDFQueueFilter)

Figure 5.6 QueueFilter and the Strategy pattern, UML

One of the behaviors of the `MemoryQueue` is that it needs to sort its `JDFQueue` in different ways. When returning the `JDFQueue` (from client call) it needs to be sorted, sometimes it needs to be returned according to a filter (in a JMF message for example). Instead of implementing these similar behaviors in the `MemoryQueue` class the strategy pattern is used to decouple the behavior (sorting, filtering) from the object itself (the `MemoryQueue` in this case).

Depending on the context, under what circumstances the `JDFQueue` is asked for, different algorithms are used to return it. It is simple to add another implementation of the `QueueFilter` and filter the `JDFQueue` at the implementers own choice. The `QueueFilter` interface is part of the Elk Framework while and the concrete classes is part of the ERI. The same principle is used for sorting a `JDFDeviceList` using the Elk `DeviceFilter` interface.

Summary: The Elk uses the Strategy pattern to decouple the behavior or the algorithms (such as sorting, filtering) of an object from the object itself. This makes it easy to extend (see Extensibility, p. 27) and to change behaviors at runtime.

5.5.4. Design pattern similarities

The terminology used in the above design patterns varies from source to source and all the patterns use a similar structure to accomplish decoupling. The most obvious differences lie in the usage of the pattern. The structure looks the same for all the above patterns. Some clarifications regarding the differences are motivated.

The difference between the Observer pattern and the Mediator pattern is that the later has a mediator class that handles the subject's observers (see Figure 5.4, p. 30). This puts less work on the subject and also makes it easy to extend and handle the registered observers in different ways e.g. by priority. It is also worth noting that the Observer pattern can also be referred to as the EventListener pattern [16]. In general, events should be quickly executed (little computational burden) and are often used when dealing with multiple threads.

In [25] the author claims that "A Command pattern is an object behavioral pattern that allows us to achieve complete decoupling between the sender and the receiver". Here, the author talks about a sender and a receiver which may be confusing since the terminology is the same for the EventListener pattern. However, the Command pattern is not used to recognize the state of an application but to execute a command, an action. In Elk's case the action is to process a JMF message.
The Strategy pattern on the other hand is used to decouple algorithms from the subject in a specific context. Different algorithms are used depending on the context. The method call to the algorithm always looks the same because the algorithms are extending the specified Strategy interface. The structure of the Strategy and the Command pattern are the same. The Strategy pattern is used to "define a family of algorithms, encapsulate each one, and make them interchangeable." [54]. It is used when the using context has some expectation about what the method will do [38]. Of course, the method calls to e.g. sort something could also be seen as an action, and the terminologies overlap again with the Command pattern. In Elk both sorting and filtering are defined by the same Strategy. It could be argued that these behaviors should be in the same "family".

5.6. Maintainability

The maintainability of Elk is achieved through a couple of generally accepted practices. The Javadoc [26] tool is using a standard way to document code including naming conventions and other important definitions when it comes to documentation. The Elk use these conventions and standards which make the code easy to understand and to maintain. Further tools that are used to make Elk easily maintained are CVS [11] and the issue tracking system JIRA [32]. JIRA is a web based tool to keep track of bugs and improvements of a project. These tools together with design principles explained in this thesis make the Elk project easily maintained.

5.7. Asynchronous processing

To start with the queue and the process need to run in different threads because their processing needs to act independently from each other. There are also a few reasons for Elk do handle asynchronous processing. Firstly, the `SubscriptionManager` needs to dispatch its internal messages asynchronously because it needs information from the `SubscribingIncomingJMFDispatcher` to be able broadcast Signals to its subscribers. The initial implementation of the `SubscriptionManager` where synchronous processing was used sometimes resulted in a deadlock.

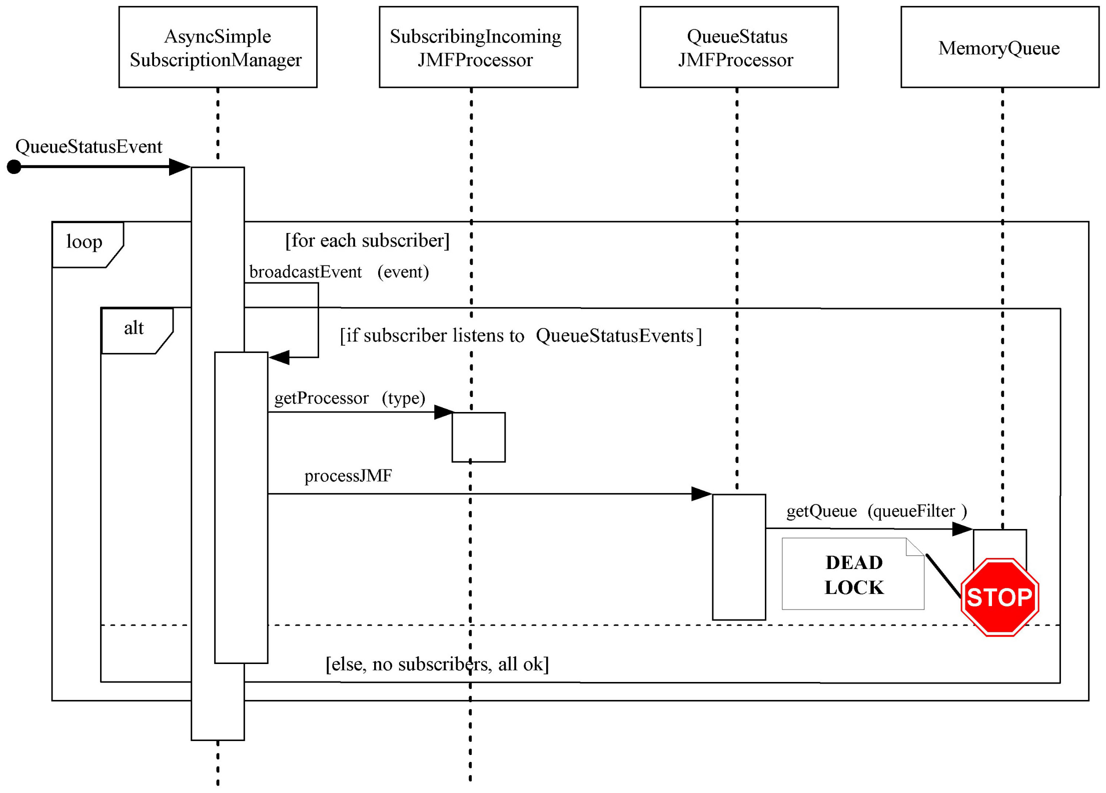


Figure 5.7 Subscription deadlock sequence diagram, UML

The `MemoryQueue` changes its status for some reason. This results in a QueueStatusEvent which can be seen in the Figure above. The `AsyncSimpleSubscriptionManager` (`SubManager`) is a listener to `QueueStatusEvents`. The `SubManager` broadcasts the event to its subscribers and in case there are any subscribers for `QueueStatusEvents` the call will result in a deadlock since the `MemoryQueue` is waiting for a return from the `SubManager`. The current implementation instead let the `SubManager` broadcast the event asynchronously (unlike the Figure above, instead the `broadCastEvent (event)` call is done asynchronously) and can return and unlock the `MemoryQueue`. The second branch of the alt box (in figure above), below the dotted horizontal line (conditional statement) applies to all other messages and results in a successful return to the `MemoryQueue` without a deadlock. The fact that events should be processed quickly is also a reason to free the `MemoryQueue` quickly using asynchronous calls.

The next usage of asynchronous processing is the `OutgoingJMFDispatcher` where the JMF messages to external users are sent. It is not desirable to have to wait for a response from an external user since the delay may be time consuming.

The JDF 1.2 Specification [27] also defines mechanism for JMF Messages to be sent asynchronously. If a JMF message includes the attribute `AcknowledgeURL` attribute the Worker (e.g. Elk) needs to respond quickly and confirm that the message was received. The actual processing is then carried out and when it is finished an Acknowledge message is sent to the URL given in the `AcknowledgeURL` of the originally received message. Some commands may be time consuming and one example of that is when a SubmitQueueEntry command is received, this is further explained in 5.8 Preprocessing on p. 35.

5.7.1. Implementation of asynchronous processing

There are several ways to implement asynchronous processing in a Web application. For Elk the JMS [23] was looked upon as well as the open source framework Mule [44]. Mule is designed around the Enterprise Service Bus (ESB) [15] architecture. One of its advantages is that it “abstracts the transport technology away from the business objects used to receive messages from the bus”. The configuration of the messaging objects are made through an xml file and is easily instantiated and used without almost any code in the business objects. Furthermore, it is conformed to work with the Spring framework. An early version of Elk used Mule to handle the asynchronous processing. However, even though Mule is a lightweight framework it turned out to add more complexity and weight than necessary. The ESB architecture is designed to handle a great amount of multiple users which is not the case for Elk.

The current implementation instead uses the util.concurrent v. 1.3.4 [46] package. The "package provides standardized, efficient versions of utility classes commonly encountered in concurrent Java programming.”. It turned out to be simple, reliable and it does fulfill the needs of the Elk asynchronous processing mechanisms.

5.8. Preprocessing

This section will explain how the Elk Reference Implementation does preprocessing on submitted JDF Nodes and the reasons for it. Below is a sequence diagram that shows the sequence of events when a JDF Node is received through a SubmitQueueEntry command JMF message.

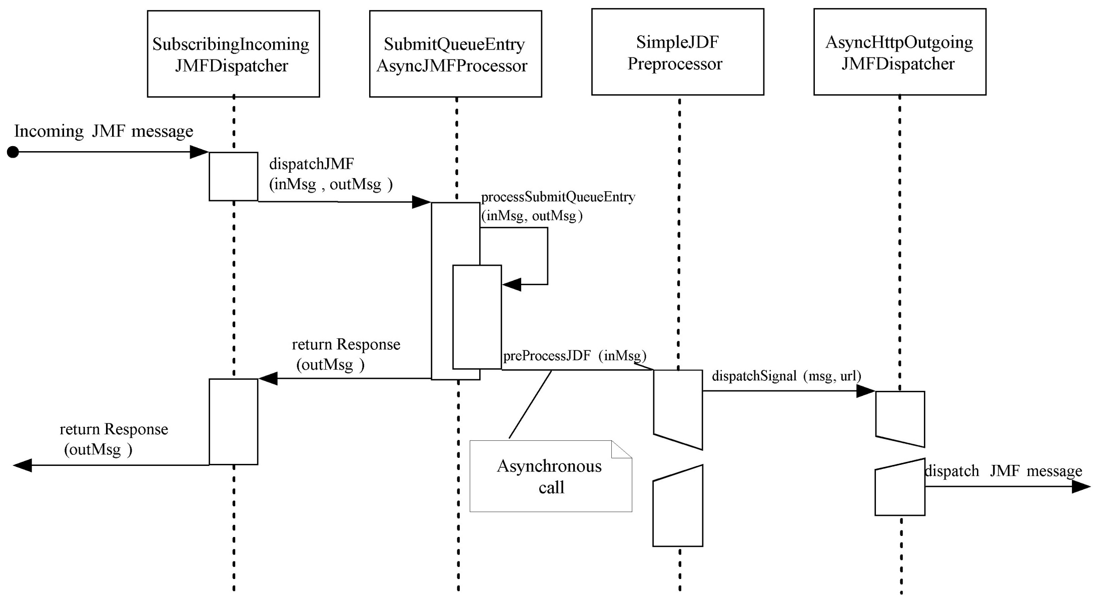

SubscribingIncoming JMFDispatcher
SubmitQueueEntry AsyncJMFProcessor
SimpleJDF Preprocessor
AsyncHttpOutgoing JMFDispatcher
Incoming JMF message
dispatchJMF (inMsg , outMsg)
processSubmitQueueEntry (inMsg, outMsg)
return Response (outMsg)
preProcessJDF (inMsg)
dispatchSignal (msg, url)
return Response (outMsg)
Asynchronous call
dispatch JMF message

Figure 5.8 Sequence diagram of a SubmitQueueEntry command

An arbitrary Invoker sends an http request and it arrives to the `DispatchingJMFServlet` which strips the contents of the request contents and forwards it to the `SubscribingIncomingJMFDispatcher` (`IncomingDispatcher`) (see figure above). The `IncomingDispatcher` will recognize the message type (a SubmitQueueEntry command) and forward it to the `SumbitQueueEntryAsyncJMFProcessor` (`SQEJMFProcessor`). The `SQEJMFProcessor` will perform one out two possible actions depending on the contents of the message. If the `AcknowledgeURL` attribute is set to a valid URL, an instant response is sent back and the preprocessing is done asynchronously (the case for the figure). If the `AcknowledgeURL` is not set, preprocessing is carried out synchronously and the `SQEJMFProcessor` can not receive new calls until the preprocessing is finished. This Invoker may have to wait a long time for the Response and there might be time outs. The preprocessor carries out a set of actions (See Figure 5.9 below) on the JDF Node and place it in the queue if the preprocessing was successful. Either way a Response (alternatively an Acknowledge) message is sent back to the Invoker (or another given URL) with appropriate Response information. The primary advantage of doing preprocessing on the JDF Node is to ensure that only correctly specified and JDF Nodes that the Device can handle and execute end up in the queue.

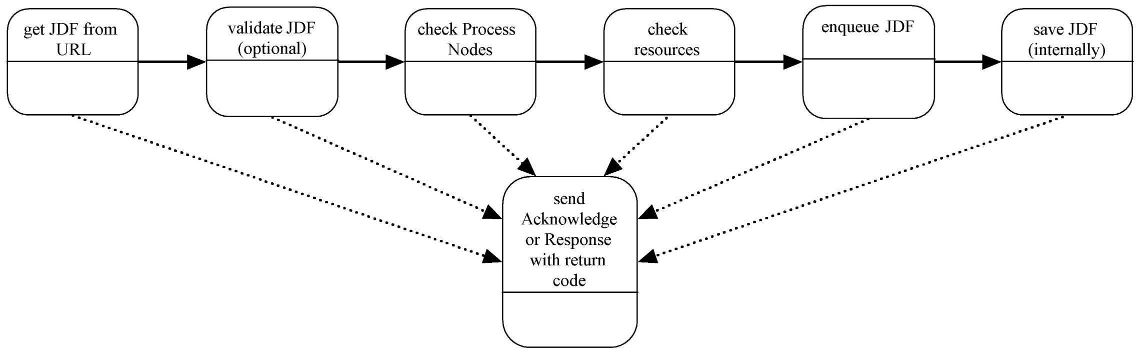


Figure 5.9 JDF Preprocessing steps

6. Discussion

The printing industry is forced to efficiently handle printed media. A graphical arts industry product is subject to change everyday and individually printed products are not unusual. One of the goals of CIP4's work is to provide the industry with tools and software applications that solves the integration of processes in a print production workflow. The JDF Specification consists of a definition of these processes in a XML-based format as well as a definition of the communication format called JMF.

The need to simplify for vendors is of great importance to make the JDF standard have the impact on the printing industry that is necessary. Single devices should be easy to replace and update without having to modify the existing software solution or the MIS. A well defined standard and a nicely designed software interface make these changes easy. To further be able to simulate and test what impact a particular change in the workflow brings more advantages to the vendors.

The Elk project is one of the tools that are provided from CIP4. Some of the purposes of Elk are:

- Give an idea on how a JDF enabled software can be designed.
- Give a concrete example of how Elk can be used to simulate a process in a JDF enabled workflow.
- Provide a framework which can be used and extended by independent vendors.
- Provide basic functionality for JMF messaging.
- Provide basic functionality for Subscription management.

The resulting software is easy to maintain and further develop because of the practices and design approaches that were chosen. At a first glance at Elk may be difficult to grasp and to get a clear overview of the class hierarchies and how they are related. The use of the Spring container to instantiate objects may be confusing. However, the advantage of using Dependency Injection over some other pattern (e.g. Service Locator pattern) outweighs the disadvantages. It adds more control over the dependencies between objects and no overhead objects are instantiated. Another thing that may be confusing is the asynchronous processing behavior. It can be hard to debug and follow the processing in an asynchronous environment. The fact that "Asynchrony gives better responsiveness and reduces the temporal coupling but is harder to debug" [19] is of course not optimal. The complexity and the difficulties to debug are eased by using logging in a proper way.

6.1. Usage scenarios

The Elk can be used in several different ways. This section describes a few usage scenarios.

6.1.1. The Elk as a simulated process in a print production workflow

Use the Elk worker as a simulated Device in a workflow and analyze what impact it has on the workflow in whole. This is also useful to test the Manager; does the manager send correct JMF messages? Does it conform to the Manager part of the Base ICS? The ERI has a two implemented processes which are the `ConventionalPrintingProcess` and the `ApprovalProcess`. It is easy to extend Elk to handle more processes by extending the `BaseProcess` (see Figure 4.2, p. 24).

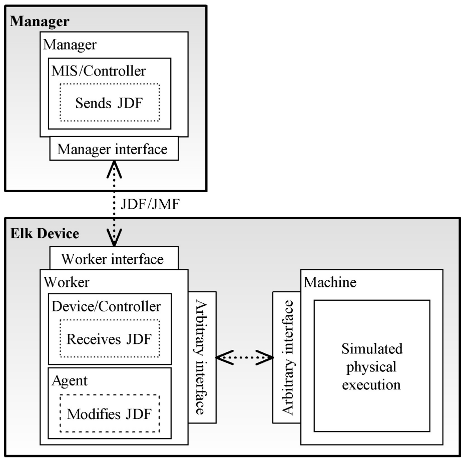


Figure 6.1 Elk as a simulated device

6.1.2. The Elk as an integration layer between a Manager and a machine

The Elk can be used as the interface towards a Manager (e.g. Alces or an MIS) where the machine interface towards Elk needs to be implemented. The implemented JMF functionality and can be used, but also extended.

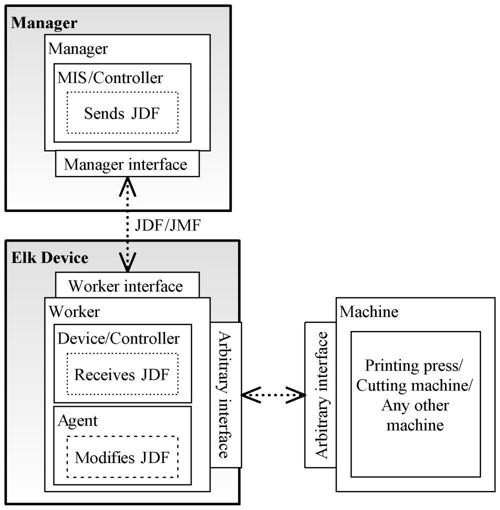


Figure 6.2 Elk as an integration layer

It is also possible to further develop and extend Elk to handle other ICS's as well as more JMF functionality or for vendor specific purposes.

6.2. Future work

The Elk is under constant development. Work on an improved version of the ConventionalPrinting has been conducted by Marco Kornrumpf. There is yet another version of the ConventionalPrinting process that is being developed. Work has also been started on a Generic process which can handle JDF Nodes of several different Process Node `Type`s.

Some of the more hands on future development areas would be to:

- Implement a persistent queue
- Implement persistent subscriptions
- Develop elk to be MIS ICS [43] conformant and to conform to other ISC's as well.
- Implement more simulated processes
- Enhance the event handling for Error events in the Elk Device
- Further develop Helk, the user interface of Elk
- …

6.3. Related work

A closely related project that implements the role of a Manager is the work on Alces, a project which was also founded by Claes Buckwalter. An add-on to Elk which also evolves with the project is Helk. Other software, such as JDFLib-J [30] and the

JDFEditor [29] are also tools that are closely related to the development work of Elk. Michael Bergman has analyzed different design approaches implementing JDF into an MIS in [6].

6.3.1. Alces

Alces [2] plays the role of a Manager (see Roles, p. 13) and is used for testing the JDF compliance of a Worker, such as a RIP, a printing press, or a binding machine. Elk is using Alces to do black box testing for Elk and comes with a pre set test suite of JMF-messages. The responses from any Worker are analyzed and validated (optional) for correctness and the results are stored in XML test report.

6.3.2. Helk

Helk is part of the Elk Project and its aim is to make Elk configurable and usable through a web interface. View and modify the queue, the jobs and the subscriptions, start and stop the process are some of the use cases of Helk. Much work has to be done (see 6.2 Future work, p. 38) for Helk to be completed.

7. Conclusion

This thesis aims to explain the functionality and design of the Elk project. Elk is a fully functional JDF Device conforming to the Base ICS. The framework strives to define the functionality needed to fulfill the job of a Worker in a print production workflow using JDF. It is developed in Java and is an open source project. This means that functionality is constantly added and the software evolves continuously. A running Elk device can be used to simulate any process of a workflow. Any application capable of sending JMF messages can use Elk as a component of their workflow.

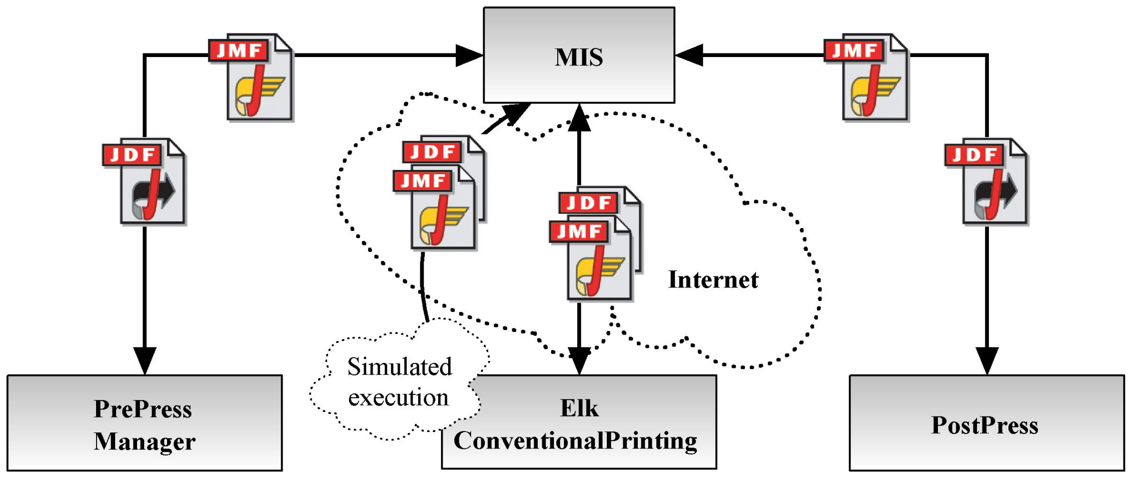


Figure 7.1 Interaction of Elk ConventionalPrinting process

This thesis also presented the architecture of the Elk project and aims to help anyone wishing to extend or use Elk in their print production workflow.

The Elk is used by different print related companies over the world as described in Usage scenarios. Hewlett Packard has implemented Elk in their own software and developed it to fit their purpose.

8. Acknowledgements

I wish to give my warm gratitude and thanks to Claes Buckwalter for excellent supervising. Also, Rainer Prosi and Dietriech Mucha have been of great help in the work with and problems arisen working with the JDF library. The members of the CIP4 Tools and Infrastructure group who inspired and listened during development of Elk should also be thanked. The experience to work in such a group meant a lot to me. Additionally, I would like to thank my colleges at ITN, LiU Norrköping, Sweden and at Heidelberger Druckmashinen AG, Kiel, Germany. I would also like to thank my room mates in Kiel who were there when I was not working and also my friends and family back home.

9. Appendix A: Running Elk

The ERI is built using an Ant build script (see Apache Ant) which contains tasks to compile, test and package the application into a WAR-file (see WAR-files). The web server that Elk has been tested on is the open source Apache Tomcat [4] server. In order to run the application the WAR-file needs to be deployed on an active server. Once the reference device is up and running it can act as an actual device in a print production workflow, and the application can receive and process jobs as MIME-packages or referenced in a JMF Message. More information on Elk; the source code and news can be found at the project web site [13]. The following sections explain the Dependencies and the instantiation of objects in Elk.

9.1. Dependencies

Of course a framework should be dependent on as little external resources as possible. However, already implemented functionality and broadly accepted frameworks and implementations can be used to reduce the burden of reinventing the wheel for each programming task. The Elk Framework depends on a few external resources which are needed in order to compile the source code and create a distribution. Some of the dependencies are:

- The Spring Framework [53], for handling the dependency injection and initialization of the reference application.
- JDFLib-J [30], all the methods and classes that are JDF specific.
- Xerces [58], the XML-parser that is used to parse the JDF and JMF instances.
- JDom [31] to store and manipulate the parsed XML-files.

External resources are also used to handle logging, http and networking, concurrent programming and servlets. These are all used when building Elk and are included in the project with JAR-files (JAR-files, see below). In order to make the Elk easily deployed into a web application it can be archived into WAR-file (see below). To easily compile and bundle the project into JARs and WARs the project’s Ant script (see Apache Ant, p. 43) is used. The WAR-file keep the internal file structure intact and make Elk easy to move, export and deploy.

9.1.1. JAR-files

The Java Archive (JAR) [47] file format enables the user to bundle multiple files into a single archive file. Most commonly a JAR file includes the necessary class files and other resources needed to compile and use an application. Additionally, the archive contains a meta-data file, a so called manifest file with versioning and creator information. Some of the benefits of having a project’s dependencies in JAR-files include:

- Security, the JAR-files can be digitally signed.
- Decreased download time. Since the files are bundled and compressed only a single download is needed.
- Versioning. A JAR file can contain version and vendor specific information.
- Portability. The mechanism for handling JAR files is a standard part of the Java platform’s core API, thus portable.

9.1.2. WAR-files

The Web Application Archives (WAR) [57] is used to store class files and information about web applications. It may contain Web components, static html pages, JSP-pages, server side and client side utility classes. A WAR has a specific directory structure. The top-level directory of a WAR is the *document root* of the application. The document root is where JSP pages, client-side classes and archives, and static Web resources are stored. The WAR-file always contains a subdirectory called WEB-INF containing deployment specific information.

9.2. Dependency injection

The Elk instantiates its objects using dependency injection using the Spring framework [53]. Using the Dependeny Injection pattern [20] decouples concrete implementations from the classes that use it. Spring is the container which is responsible of instantiating and injecting the necessary dependencies to the objects. In Elk many of the classes are Singletons and it is an advantage to take care of the instantiations at a single location: the Spring container. The Spring container is configured in a XML file and the implementing classes can easily be exchanged.

9.2.1. Spring in Elk

The dependencies (between object) in Elk are mainly injected through the constructor (so called constructor-injection). E.g. the `ConventialPrintingProcess` class is dependent on a `Config` object (to be able to get the device's configuration and capabilities), a `Queue` (to get jobs from the queue), `URLAccessTool` (to access the file system in a convenient way), a `Repository` (to access locally stored JDF's). Instead of having the `ConventionalPrintingProcess` creating these objects they are passed as arguments to the constructor. The Spring container take care of the instantiation. Particularly, this is done during startup of the device and it is configured in the `elk-spring-config.xml` file. This file is located in the `WEB-INF/classes` directory of the ConventinalPrinting device. All classes are defined here using the Enterprise Java Beans (EJB) [14] principle, the example above looks like this:

```
<bean id="process"
class="org.cip4.elk.impl.device.process.ConventionalPrintingProcess"
singleton="true" init-method="init">
            <constructor-arg>
                  <ref bean="deviceConfig"/>
            </constructor-arg>
            <constructor-arg>
                  <ref bean="queue"/>
            </constructor-arg>
            <constructor-arg>
                  <ref bean="fileUtil"/>
            </constructor-arg>
            <constructor-arg>
                  <ref bean="outgoingDispatcher"/>
            </constructor-arg>
            <constructor-arg>
                  <ref bean="fileRepository"/>
            </constructor-arg>
            <property name="incomingDispatcher">
                  <ref bean="incomingDispatcher"/>
            </property>
</bean>
```

The `class` attribute of the bean tells Spring which class to instantiate.

9.3. Apache Ant

Apache Ant [3] is java-based build tool. The configuration is made in a simple XML-based format and it can easily be extended and application specific *tasks* can be created. The ant build script (usually in file called `build.xml`) contains tasks that perform certain commands on the project being built. Clean, compile are probably the most common tasks used in ant. The Elk uses plugged in tasks to start, restart and deploy the application into the Jakarta Tomcat server.

In Elk, the ant script is used for the following tasks:

- Compiling the source code.
- Generating JAR-file.
- Cleaning the output directories.
- Autorun the unit tests.
- Controlling the Jakarta Tomcat [4] server.
- Generate WAR-files for the implemented processes.
- Compile a program that can send JMF messages (used for testing)

9.3.1. Usage of Ant in Elk

There are several ways to perform the tasks above. From an IDE, such as Eclipse [12], the Ant tasks can automatically be displayed from the IDE environment. The `build.xml` file is placed in the project root folder. From a command line console all that is necessary to do is to write `ant` to execute the default task (compile, bundle into jar-files and war-files). To see what other tasks there are type `ant -p`.

10. Appendix B: Glossary

10.1. Software related terms

Alces	Alces plays the role of a Manager and is used for testing the JDF compliance of a Worker, such as a RIP, a printing press, or a binding machine. (see 6.3.1 Alces, p. 39)
Command Pattern	A Command pattern is an object behavioral pattern that allows us to achieve complete decoupling between the sender and the receiver (see Command Pattern, p. 31)
Decoupling	The practice to make objects independent from each other. (see Decoupling, p. 29)
Elk project	The work of developing the Elk Framework and the Elk Reference Implementation. The project's web site can be found at [13].
Enterprise Service Bus (ESB)	Enterprise service bus is a generic name for any solution that provides communication and translation between applications or between different processes within an application [15].
Framework	A framework is a reusable design consisting of abstract classes and interfaces.
Helk	Helk is part of the Elk Project and its aim is to make Elk configurable and usable through a web interface. (see 6.3.2 Helk, p. 39)
Regression testing	The practice to run the same tests regularly to ensure the correctness of the software remains.

Refactoring	Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior (see Refactoring, p. 28).
Servlet	Servlets are the Java platform technology of choice for extending and enhancing Web servers [24].
Spring Framework	The Spring framework is a wide ranging framework for enterprise Java development. (see 9.2 Dependency injection, p. 42)
The Elk Framework	An application framework that provides the services needed by a JDF Device or Controller. The Elk Framework API (see 3 The Elk Framework, p. 17) is developed under the Elk project.
The Elk Reference Implementation (ERI)	The Elk Reference Implementation (see 4 The Elk Reference Implementation) is an implementation of the classes specified by the Elk Framework. The ERI can also be referred to as the Elk Device and has the role of a Worker.
XML (eXtensible Markup Language)	XML is a markup language which defines a way to describe data according to a standardized set of rules.

10.2. Print Related terms

Agent	An agent can write, create and modify JDF Nodes. Anything that can be used to create a JDF Node is an agent, for example a simple text editor. (see 2.2.3 Roles, p. 13)
Approval	The process of approving a media in order to let it further in the workflow. This process is commonly done by person; a customer or an employee.(see 4.2.1 ApprovalProcess, p. 23)

Binding	The processes involved in combining multiple products, bind them together. Examples from the JDF Specification are *AdhesiveBinding*, *ChannelBinding*, and *CoilBinding*.
BoxPacking	The process of packing a stack, bundle or pile of a product into a box or a cartoon [27].
ColorCorrection	ColorCorrection is the process of modifying the specification of colors in documents to achieve some desired visual result [27].
Controller	A Controller can split up and route JDF Nodes to appropriate Devices. A controller must at least be able to initiate one process, controller or device. (see 2.2.3 Roles, p. 13)
ConventionalPrinting	This process covers several conventional printing tasks, including sheetfed printing, web printing, web/ribbon coating, converting, and varnishing [27]. (see 4.2.2 ConventionalPrinting, p. 23)
Device	A Device is capable of executing the information it receives from an *Agent* or a *Controller*. Devices must be able to execute JDF Nodes and initiate its underlying machine. (see 2.2.3 Roles, p. 13)
Machine	A machine is any part of the workflow system designed to execute a process. Most often, this term refers to a piece of physical equipment, such as a press or a binder, but it can also refer to the software components used to run a particular machine. (see 2.2.3 Roles, p. 13)

Management Information System (MIS)	The *Management Information System* is controlling and overseeing the complete workflow and its units. It is also responsible for monitoring the progress of the job. The MIS can do this via JMF or by information from the JDF Nodes audits. (see 2.2.3 Roles, p. 13)
Manager	The software that implements the Manager Interface. (see 2.2.3 Roles, p. 13)
Manager Interface	The interface that sends JDF Instances and JMF messages to a Worker in a Device or a Controller. (see 2.2.3 Roles, p. 13)
Plate making	The process of making the plates ready to be used in a printing press.
Postpress	The process involved after printing such as packing, laminating, and hole making.
Preflighting	Preflighting is the process of examining the components of a print job to ensure that the job will print successfully and with the expected results [27].
Prepress	The processes involved before the media goes to print. Examples are ColorCorrection, Scanning, and Preflighting.
Rendering	Rendering is the process of generating an image from a model, by means of a software program. [49]
RIPing (Raster Image Processing)	The process of rasterizing pages before going to print.
Worker	The software that implements the Worker Interface. (see 2.2.3 Roles, p. 13)
Worker Interface	The interface that receives JDF Instances and JMF messages from a Manager in a Controller or a MIS. (see 2.2.3 Roles, p. 13)

11. Appendix C: The author's programming contributions

This appendix explains in more detail what programming tasks have been solved by the author.

I have only contributed to the project's conformance to the Base ICS. I have not invented the overall design; this was made by Claes Buckwalter.

For each task a reference to a package is given where the task was implemented. The amount of code I have contributed to the project can be found in Appendix D: Code statistics for Elk.

11.1. KnownDevices Message

Code in package: `org.cip4.elk.impl.jmf`
The KnownDevices message is sent to a worker (ERI) to get information about the device, what tasks it can perform and which processes it handles. It may also return information of the Device's current state. If the device is currently processing a job the JobPhase element is also returned. The JobPhase contains information of the status of the device, such as a percentage of the total amount of sheets to print. For the KnownDevices message to conform to Base ICS, I had to implement the DeviceFilter attribute from the JDF Specification, which specifies what information should be returned.
Problems to solve:

How do I get the status of the Device (i.e the JobPhase element)?
How do I implement a DeviceFilter in an object oriented way?

Since Elk is a multi-threaded application and the Process is executing its jobs in a separate thread I had to include the Process in the constructor of the KnownDevices class. The process is injected during startup of the application by the Spring container. Furthermore I needed to implement methods for accessing the JobPhase element of `org.elk.impl.device.process` package.

I implemented a `BaseICSDeviceFilter` to filter the device information returned according to the Base ICS. The `BaseICSDeviceFilter` implemented the interface `DeviceFilter` which I also developed and which is further explained in Strategy pattern on p. 32.

11.2. JDF Preprocessing

Code in package: `org.cip4.elk.impl.jmf.preprocess`
Here the problem was to investigate the need for JDF pre processing and the actions to be taken in the preprocessing state. The reason for pre processing is to avoid incomplete, not runnable or non handled JDF jobs to end up in the device's queue.

I came up with a sequential solution where the JDF job is checked for validity in the order of that in Figure 5.9 JDF Preprocessing steps, p. 36.

11.3. Subscriptions functionality

Code in packages: `org.cip4.elk.impl.subscriptions`, `org.cip4.elk.impl.jmf.preprocess`, `org.cip4.elk.device.process`

The implementation for handling subscriptions had certain limitations before I started working on it. It only supported event-based subscriptions and did not handle subscriptions asynchronously. The subscription mechanism was implemented in the `SimpleSubscriptionManager` class. Some additional features that was required from the assigner:

- Support for Time-based Subscriptions
- Subscribing through JDF/NodeInfo
- Handle subscriptions asynchronously

In accomplishing this I developed a separate package to handle the new features, `org.cip4.elk.impl.subscriptions` (see Figure 5.1 Overview of the subscriptions package, UML, p. 28). To handle the time-based subscriptions an inner java class extending the java `TimerTask` class was developed in the `BaseICSSubscriber`.

The enabling of subscriptions through the JDF/NodeInfo element was most appropriately implemented in the `SimpleJDFPreprocessor`. The preprocesser initiates a subscription after the JDF Node had been checked with the preprocessor. The `BaseProcess` also had to be extended to be able to cancel a subscription once the processing of a node was finished.

11.4. Asynchronous processing

In order to make the Elk device work properly I had to implement asynchronous handling of incoming and outgoing JMF messages partly to prevent deadlocks. Another requirement was to handle the AcknowledgeURL attribute of a JMF/Command element. This is stated in the Base ICS documents and informs submitter of a JDF or another incoming command that the device (Elk) has received the message and that it will process it.

I investigated different ways to implement these features and looked at Sun's Java Message Serivce (JMS) API [23]. Another thing I looked at was Mule[44], which is light weight framework for asynchronous messaging. I implemented the above features using Mule, but it turned out that it added more complexity and weight than desired. The current version of Elk is using the util.concurrent v. 1.3.4 [46] package. Also see section 5.7 Asynchronous processing, p. 33 for more information.

11.5. ConventionalPrinting process

Code in package: `org.cip4.elk.impl.device.process`

Before I started to work on the Elk project only the Approval process was implemented. To make Elk more useful a simulation device of print production workflow the implementation of a printing press was a desired functionality. In implementing the ConventionalPrinting process this functionality was accomplished. A printing press produces printed sheets and the programming tasks involved with this was to implement the handling of Amount events, the subscription of Amount events. The problems arisen with this was implemented in an object oriented way earlier described in
5 Program Design, Concepts and Development on p. 25.

11.6. Additional features

A lot of other functionality was also added in order to make Elk conformant to the Base ICS. A relatively detailed specification of the code I contributed to the project can be seen in Appendix D: Code statistics for Elk, below and it can also be browsed online at [13], in the link project reports, source Xref.

Some of the additional features that was implemented:
In the `org.cip4.elk.impl.jmf package`:
KnownMessages/MsgQuParams
StopPersistentChannel to be base ICS
SubmissionMethods message
QueueEntryDetails

Other features that were implemented that involved modifications across packages were:

- Processes
  - Follows JDF/NodeInfo/TargetRoute
  - Improved Audit information
  - Bug fixes
- SubmitQueueEntries asynchronously
- AcknowledgeURL for SubmitQueueEntry.
- Automated testing.
- Refactoring, bug fixes.

These features were all solved in the object oriented fashion explained in section Program Design, Concepts and Development on p. 25.

11.7. Time consumption

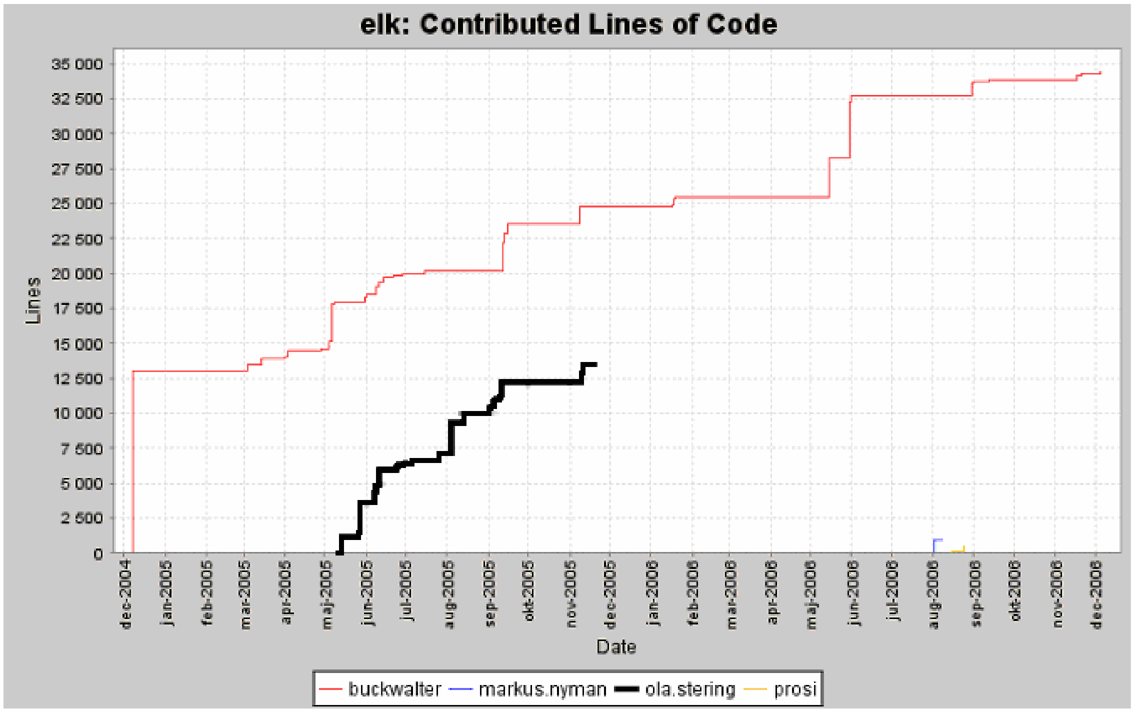
The first two-three months I was reading and learning the concepts of XML and JDF. I also spent a lot of time throughout the thesis period to learn the basics of the printing industry. I was also looking into Spring and dependency injection strategies to understand how the Elk application was run. I had to understand and learn the basics of the Apache Tomcat server, the Ant-build tools and gather knowledge in the Eclipse IDE. When implementing asynchronous processing I was looking into JMS, Mule and java.util packages. Furthermore I got acquainted with different tools for developing UML diagrams such as gentlware's Poseidon for UML [21] , Smart Draw [51] and such. A lot of the programming involved changing and expanding existing code and these improvements were carried out throughout the whole thesis period. I started to contribute to the project a little earlier than what is shown in CVS because I did not have my CVS account ready until the beginning of May. The primary programming tasks, which took about one month each were in these categories: Asynchronous programming (Mule, JMS, etc.), Subscription package, pre processing and the Conventional printing. Alongside with these the other additional features were implemented.

12. Appendix D: Code statistics for Elk

12.1. Project totals

Author	Lines of Code
buckwalter	34352 (69.7%)
ola.stering	13525 (27.4%)
markus.nyman	954 (1.9%)
prosi	444 (0.9%)

12.2. Lines of Code



12.3. Ola Stering's contributions

Elk Developers: ola.stering
Login name: ola.stering
Total Commits: 310 (29.5%)
Lines of Code: 13525 (27.4%)

12.3.1. Activity in Directories (Ola Stering)

Directory	Changes	Lines of Code	Lines per Change
Totals	310 (100.0%)	13525 (100.0%)	43.6
src/java/org/cip4/elk/impl/device/process/	32 (10.3%)	2325 (17.2%)	72.6
src/java/org/cip4/elk/impl/subscriptions/	23 (7.4%)	1768 (13.1%)	76.8
src/java/org/cip4/elk/impl/jmf/util/	8 (2.6%)	1031 (7.6%)	128.8
src/java/org/cip4/elk/impl/jmf/preprocess/	6 (1.9%)	857 (6.3%)	142.8
src/java/org/cip4/elk/impl/queue/	13 (4.2%)	753 (5.6%)	57.9
src/test/org/cip4/elk/impl/jmf/	14 (4.5%)	706 (5.2%)	50.4
src/test/data/	21 (6.8%)	555 (4.1%)	26.4
src/java/org/cip4/elk/impl/jmf/	11 (3.5%)	524 (3.9%)	47.6
src/java/org/cip4/elk/impl/servlet/	4 (1.3%)	502 (3.7%)	125.5
src/test/org/cip4/elk/impl/subscriptions/	5 (1.6%)	496 (3.7%)	99.2
src/test/org/cip4/elk/impl/device/process/	3 (1.0%)	474 (3.5%)	158.0
src/java/org/cip4/elk/device/process/	9 (2.9%)	468 (3.5%)	52.0
src/devices/ConventionalPrinting/	1 (0.3%)	388 (2.9%)	388.0
src/java/org/cip4/elk/impl/queue/util/	3 (1.0%)	302 (2.2%)	100.6
src/test/org/cip4/elk/impl/jmf/preprocess/	4 (1.3%)	298 (2.2%)	74.5
src/test/org/cip4/elk/impl/jmf/util/	4 (1.3%)	265 (2.0%)	66.2
src/test/org/cip4/elk/testtools/servlet/	1 (0.3%)	263 (1.9%)	263.0
src/test/org/cip4/elk/impl/device/	3 (1.0%)	240 (1.8%)	80.0
src/java/org/cip4/elk/impl/util/	2 (0.6%)	186 (1.4%)	93.0
src/test/org/cip4/elk/jmf/servlet/	2 (0.6%)	169 (1.2%)	84.5
src/test/org/cip4/elk/impl/queue/	7 (2.3%)	156 (1.2%)	22.2
src/java/org/cip4/elk/impl/queue/jmf/	10 (3.2%)	148 (1.1%)	14.8
src/java/org/cip4/elk/impl/device/	8 (2.6%)	126 (0.9%)	15.7
src/test/org/cip4/elk/impl/queue/util/	4 (1.3%)	110 (0.8%)	27.5
src/test/org/cip4/elk/impl/util/	2 (0.6%)	104 (0.8%)	52.0
src/java/org/cip4/elk/impl/device/jmf/	3 (1.0%)	86 (0.6%)	28.6
src/test/org/cip4/elk/	2 (0.6%)	82 (0.6%)	41.0

Directory	Changes	Lines of Code	Lines per Change
src/java/org/cip4/elk/	4 (1.3%)	71 (0.5%)	17.7
src/test/org/cip4/elk/impl/jmf/mime/	1 (0.3%)	35 (0.3%)	35.0
src/java/org/cip4/elk/queue/	5 (1.6%)	28 (0.2%)	5.6
/	2 (0.6%)	8 (0.1%)	4.0
src/java/org/cip4/elk/impl/	1 (0.3%)	1 (0.0%)	1.0
src/test/data/jdf/	91 (29.4%)	0 (0.0%)	0.0
src/devices/Approval/	1 (0.3%)	0 (0.0%)	0.0

Statistics generated by StatCVS 0.3

13. References

[1] Abstract factory pattern [Online]. Available: http://en.wikipedia.org/wiki/Abstract_factory_pattern [Sep 2006]

[2] Alces [Online]. Available: http://elk.itn.liu.se/alces/ [Sep 2006]

[3] Apache Ant [Online]. Available: http://ant.apache.org/ [Sep 2006]

[4] Apache Tomcat [Online]. Available: http://tomcat.apache.org/ [Sep 2006]

[5] Base ICS [Online]. Available: http://www.cip4.org/document_archive/documents/ICS-Base-1.0RevB.pdf [Sep 2006]

[6] Bergman, Michael. *JDF design approaches with implementation in a Management Information System*. Master of Science thesis in Information Systems and Database Technology. Royal Institute of Technology, Stockholm, Sweden, 2005.

[7] Bloch. *Effective Java, Programming Language Guide*. Addison-Wesley, 2004, ISBN: 0-201-31005-8

[8] Buckwalter, Claes. *A JDF-enabled Workflow Simulation Tool*, Proceedings TAGA 2005 Conference, Toronto, Canada.

[9] Command pattern [Online]. Available: http://en.wikipedia.org/wiki/Command_pattern [Sep 2006]

[10] Cruise Control [Online]. Available: http://cruisecontrol.sourceforge.net/ [Sep 2006]

[11] CVS [Online]. Available: http://www.gnu.org/software/cvs/ [Sep 2006]

[12] Eclipse SDK [Online]. Available: http://www.eclipse.org/ [Sep 2006]

[13] Elk [Online]. Available: http://elk.itn.liu.se/ [Sep 2006]

[14] Enterprise JavaBeans Technology [Online]. Available: http://java.sun.com/products/ejb/ [Sep 2006]

[15] Enterprise Service Bus [Online]. Available: http://en.wikipedia.org/wiki/Enterprise_Service_Bus [Sep 2006]

[16] Event listener [Online]. Available: http://en.wikipedia.org/wiki/Event_listener [Sep 2006]

[17] Extreme programming [Online]. Available: http://en.wikipedia.org/wiki/Extreme_Programming [Sep 2006]

[18] eXtreme programming [Online]. Available: http://www.extremeprogramming.org/ [Sep 2006]

[19] Fowler, Martin. UML Distilled Third Edition, a Brief Guide to the Standard Object Modeling Language. Addison-Wesley. 2004. ISBN: 0-321-19368-7

[20] Inversion of Control Containers and the Dependency Injection pattern [Online]. Available: http://www.martinfowler.com/articles/injection.html [Sep 2006]

[21] Gentlware, Poseidon for UML [online]. Available: http://www.gentleware.com/ [March 2007]

[22] Iterative and incremental development [Online]. Available: http://en.wikipedia.org/wiki/Iterative_development [Sep 2006]

[23] Java Message Service (JMS) [Online]. Available: http://java.sun.com/products/jms/ [Sep 2006]

[24] Java Servlet Technology [Online]. Available: http://java.sun.com/products/servlet/overview.html [Sep 2006]

[25] Java Tip 68: Learn how to implement the Command pattern in Java [Online]. Available: http://www.javaworld.com/javaworld/javatips/jw-javatip68.html [Sep 2006]

[26] Javadoc Tool [Online]. Available: http://java.sun.com/j2se/javadoc/ [Sep 2006]

[27] JDF 1.2 Specification [Online]. Available: http://www.cip4.org/documents/jdf_specifications/JDF1.2.pdf [Sep 2006]

[28] JDF ICS documents [Online]. Available: http://www.cip4.org/document_archive/ics.php [Sep 2006]

[29] JDF Windows Editor [Online]. Available: http://www.cip4.org/open_source/jdfeditor-2.1.3.36-win.zip [Sep 2006]

[30] JDFLib-J API [Online]. Available: http://www.cip4.org/open_source/doc/jdf_java/ [Sep 2006]

[31] JDom [Online]. Available: http://www.jdom.org/ [Sep 2006]

[32] JIRA [Online]. Available: http://www.atlassian.com/software/jira/ [Sep 2006]

[33] Johansson, Lundberg, Ryberg , *Grafisk kokbok 2.0*, FörlagKOKO, ISBN 9178431611

[34] JUnit [Online]. Available: http://www.junit.org/index.htm [Sep 2006]

[35] JUnit Test Infected: Programmers Love Writing Tests [Online]. Available: http://junit.sourceforge.net/doc/testinfected/testing.htm [Sep 2006]

[36] Kernighan, Brian W, Pike, Rob. *The practice of programming*. Addison-Wesley. 1999. ISBN: 0-201-61586-9

[37] Liljegren, Gustav. *XML – begreppen och tekniken*, Studentlitteratur 2004, ISBN 9144024762

[38] Liskow, Guttag. *Program Development in Java*. Addison-Wesely, 2001, ISBN: 0-201-65768-6.

[39] Log4j project [Online]. Available: http://logging.apache.org/log4j/docs/ [Sep 2006]

[40] Logging in Java Applications [Online]. Available: http://www.developer.com/java/other/article.php/1404951 [Sep 2006]

[41] McLaughlin, *Java & XML*, O'Reilly Media, 2001, ISBN 0596001975

[42] Medbo, Anders. *How to become profitable with JDF*. Presentation. Grafex 2005.

[43] MIS ICS [Online]. Available: http://www.cip4.org/document_archive/documents/ICS-MIS-1.0RevA.pdf [Sep 2006]

[44] Mule [Online]. Available: http://mule.codehaus.org/ [Sep 2006]

[45] Observer pattern [Online]. Available: http://en.wikipedia.org/wiki/Observer_pattern [Sep 2006]

[46] Overview of package util.concurrent Release 1.3.4. [Online]. Available: http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html [Sep 2006]

[47] Packaging Programs in JAR Files [Online]. Available: http://java.sun.com/docs/books/tutorial/deployment/jar/ [Sep 2006]

[48] Refactroing [Online]. Available: http://www.refactoring.com/ [Sep 2006]

[49] Rendering [Online]. Available: http://en.wikipedia.org/wiki/Rendering_(computer_graphics) [Sep 2006]

[50] Singleton pattern [Online]. Available: http://en.wikipedia.org/wiki/Singleton_pattern [Sep 2006]

[51] Smartdraw. [Online]. Available: http://www.smartdraw.com/ [March 2007]

[52] Software testing [Online]. Available: http://en.wikipedia.org/wiki/Software_testing [Sep 2006]

[53] Spring Framework [Online]. Available: http://www.springframework.org/ [Sep 2006]

[54] Strategy for success [Online]. Available: http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html [Sep 2006]

[55] Strategy pattern [Online]. Available: http://en.wikipedia.org/wiki/Command_pattern [Sep 2006]

[56] The International Cooperation for the Integration of Processes in Prepress, Press and Postpress (CIP4) [Online]. Available: http://www.cip4.org [Sep 2006]

[57] Web Application Archives [Online]. Available: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html [Sep 2006]

[58] Xerces [Online]. Available: http://xerces.apache.org/ [Sep 2006]